ADSP-BF54x Blackfin® Processor Hardware Reference

(includes ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, ADSP-BF549)

Revision 1.2, February 2013

Part Number
82-100108-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

CONTENTS

PREFACE

Purpose of This Manual lxxxi
Intended Audience lxxxi
What's New in This Manual lxxxii
Technical Support lxxxiii
Supported Processors lxxxiv
Product Information lxxxv
Analog Devices Web Site lxxxv
EngineerZone lxxxvi
Notation Conventions lxxxvii
Register Diagram Conventions lxxxviii

INTRODUCTION

Peripherals 1-2
Memory Architecture 1-5
Internal Memory 1-6

External Memory 1-7
NAND Flash Controller (NFC) 1-8
I/O Memory Space 1-9
One-Time-Programmable (OTP) Memory 1-9
DMA Support 1-10
Host DMA Interface 1-12
External Bus Interface Unit 1-13
DDR SDRAM Controller 1-13
Asynchronous Controller 1-14
Ports 1-14
General-Purpose I/O (GPIO) 1-14
Two-Wire Interface 1-15
Controller Area Network 1-16
Enhanced Parallel Peripheral Interface (EPPI) 1-17
SPORT Controllers 1-19
Serial Peripheral Interface (SPI) Port 1-20
Timers 1-21
UART Ports 1-22
USB On-The-Go, Dual-Role Device Controller 1-23
ATA/ATAPI–6 Interface 1-24
Keypad Interface 1-24
Secure Digital (SD)/SDIO Controller 1-25
Rotary Counter Interface 1-26
Security 1-26

Media Transceiver Mac Layer (MXVR) 1-27
Real-Time Clock 1-29
Watchdog Timer 1-30
Clock Signals 1-30
Dynamic Power Management 1-31
Full On Mode (Maximum Performance) 1-31
Active Mode (Moderate Dynamic Power Savings) 1-31
Sleep Mode (High Dynamic Power Savings) 1-32
Deep Sleep Mode (Maximum Dynamic Power Savings) 1-32
Hibernate State (Maximum Power Savings) 1-33
Voltage Regulation 1-33
Boot Modes 1-33
Instruction Set Description 1-34
Development Tools 1-36

CHIP BUS HIERARCHY

Overview 2-1
Internal Interfaces 2-1
Internal Clocks 2-5
Core Bus Overview 2-6
System Overview 2-8
P Port Interface 2-8
D Port Interface 2-9
On-Chip L2 Interface 2-11

Peripheral Access Bus (PAB) 2-15
PAB Performance 2-15
PAB Agents (Masters, Slaves) 2-15
DMA-Related Buses 2-17
Peripheral DMA 2-18
DAB Bus Agents (Masters) 2-18
DAB Arbitration 2-18
DCB Arbitration 2-20
DEB Arbitration 2-22
DAB, DCB, and DEB Performance 2-22
External Access Bus (EAB) 2-24
EAB/DEB Arbitration 2-25
EAB/DEB Performance 2-25

MEMORY

Memory Architecture 3-2
Internal Memory 3-5
Overview of L1 Instruction SRAM 3-6
Overview of L1 Instruction ROM 3-6
Overview of L1 Data SRAM 3-7
Overview of Scratchpad Data SRAM 3-7
Overview of On-Chip L2 3-8

L1 Instruction Memory 3-8
Instruction Memory Control Register (IMEM_CONTROL) 3-9
L1 Instruction SRAM 3-11
L1 Instruction Cache 3-13
Cache Lines 3-13
Cache Hits and Misses 3-17
Cache-Line Fills 3-17
Line-Fill Buffer 3-18
Cache-Line Replacement 3-18
Instruction Cache Management 3-20
Instruction Cache Locking by Line 3-20
Instruction Cache Locking by Way 3-21
Instruction Cache Invalidation 3-22
Instruction Test Registers 3-23
ITEST_COMMAND Register 3-24
ITEST_DATA1 Register 3-25
ITEST_DATA0 Register 3-26
L1 Data Memory 3-27
Data Memory Control Register (DMEM_CONTROL) 3-27
L1 Data SRAM 3-30
L1 Data Cache 3-33
Example of Mapping Cacheable Address Space into Data Banks 3-34
Data Cache Access 3-37

Cache Write Method 3-39
Write Buffers 3-39
Interrupt Priority Register (IPRIO) and Write Buffer Depth 3-40
Data Cache Control Instructions 3-41
Data Cache Invalidation 3-42
Data Test Registers 3-42
Data Test Command Register (DTEST_COMMAND) 3-44
Data Test Data 1 Register (DTEST_DATA1) 3-45
Data Test Data 0 Register (DTEST_DATA0) 3-46
On-Chip Level 2 (L2) Memory 3-47
On-Chip L2 Bank Access 3-47
Latency 3-48
One Time Programmable Memory 3-49
External Memory 3-50
Memory Protection and Properties 3-51
Memory Management Unit 3-51
Memory Pages 3-53
Memory Page Attributes 3-53
Page Descriptor Table 3-54
CPLB Management 3-55
MMU Application 3-56
Examples of Protected Memory Regions 3-58
ICPLB Data Registers (ICPLB_DATAx) 3-59
DCPLB Data Registers (DCPLB_DATAx) 3-61

DCPLB Address Registers (DCPLB_ADDRx) 3-63
ICPLB Address Registers (ICPLB_ADDRx) 3-64
CPLB Status Registers 3-65
DCPLB Status Register (DCPLB_STATUS) 3-66
ICPLB Status Register (ICPLB_STATUS) 3-66
CPLB Fault Address Registers 3-67
DCPLB Fault Address Register (DCPLB_FAULT_ADDR) 3-68
ICPLB Fault Address Register (ICPLB_FAULT_ADDR) 3-69
Memory Transaction Model 3-69
Load/Store Operation 3-70
Interlocked Pipeline 3-71
Ordering of Loads and Stores 3-72
Synchronizing Instructions 3-73
Speculative Load Execution 3-74
Conditional Load Behavior 3-75
Working With Memory 3-76
Alignment 3-76
Cache Coherency 3-76
Atomic Operations 3-77
Memory-Mapped Registers 3-78
Core MMR Programming Code Example 3-78
Terminology 3-79

ONE-TIME PROGRAMMABLE MEMORY

OTP Memory Overview 4-1
OTP Memory Map 4-2
Error Correction 4-6
Error Correction Policy 4-6
OTP Access 4-8
OTP Timing Parameters 4-10
OTP Timing Calculations for SCLK = 100 MHz 4-11
OTP Timing Calculations for SCLK = 50 MHz 4-12
OTP Timing Calculations for SCLK = 40 MHz 4-12
OTP_TIMING Register 4-13
Callable ROM Functions for OTP ACCESS 4-14
Initializing OTP 4-14
bfrom_OtpCommand 4-14
Programming and Reading OTP 4-16
bfrom_OtpRead 4-17
bfrom_OtpWrite 4-18
Error Codes 4-21
Write Protecting OTP Memory 4-24
Accessing Private OTP Memory 4-26
OTP Programming Examples 4-26
Enable Access to Private OTP 4-27
Enable Access to Private OTP and Enable JTAG Emulation in Secure Mode 4-27

Read Public OTP Memory and Print to Console 4-27
OTP Write to Single Page Using Two Half Page Accesses 4-29
Lock Page Without Writing Any Data 4-30

EXTERNAL BUS INTERFACE UNIT

General Overview 5-2
Block Diagram 5-4
On-Chip System Interfaces 5-7
Error Detection 5-8
System Arbitration 5-8
Address Resolution 5-9
Reorder Unit 5-9
DDR Queue Manager 5-11
DDR Arbitration 5-11
DDR SDRAM Controller 5-15
Features 5-15
DDR SDRAM Controller 5-15
Mobile DDR SDRAM Controller 5-16
Partial Array Self-refresh 5-16
Memory Driver Strength 5-17
Temperature Compensated self-refresh 5-17

Unsupported Mobile DDR SDRAM Controller Features ... 5-17
Deep Power Down 5-17
Clock Stop Mode 5-17
Clock Frequency During Operation 5-18
DDR SDRAM Memory Interface 5-18
DDR SDRAM Programming Model 5-19
Recommended Programming Sequence 5-21
DDR Registers 5-23
Memory Control Register 0 (EBIU_DDRCTL0) 5-24
Memory Control Register 1 (EBIU_DDRCTL1) 5-25
Memory Control Register 2 (EBIU_DDRCTL2) 5-26
Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices 5-27
Memory Control Register 3 (EBIU_DDRCTL3), Mobile DDR Devices 5-28
Queue Configuration Register (EBIU_DDRQUE) 5-29
Reset Control Register (EBIU_RSTCTL) 5-30
Error Master Register (EBIU_ERRMST) 5-31
Error Address Register (EBIU_ERRADD) 5-32
Mode of Operation - DDR 5-32
Data Flow for 16-bit DDR SDRAMs 5-33
Definition of Standard DDR Terms 5-34
DDR SDRAM System Organization 5-40
DDR SDRAM Configurations Supported 5-41
DDR Timing Parameter Definitions 5-43

DDR Metrics Control Registers 5-44
DDR Metrics Counter Enable (EBIU_DDRMCEN) Register 5-44
DDR Metrics Counter Clear (EBIU_DDRMCCL) Register 5-47
DDR READ Access Count (EBIU_DDRBRCx) Registers 5-49
DDR WRITE Access Count (EBIU_DDRBWCx) Registers 5-50
DDR Page ACTIVATE Count (EBIU_DDRACCT) Register 5-51
DDR TURN AROUND Count (EBIU_DDRTACT) Register 5-51
DDR AUTO-REFRESH Count (EBIU_DDRARCT) Register 5-51
DDR Grant Count (EBIU_DDRGCx) Registers 5-51
More Grant Counter Options 5-52
DDR Grant Count Control 5-53
Asynchronous Memory Interface 5-53
Asynchronous Memory Address Decode 5-54
Asynchronous Memory Arbitration 5-55
Asynchronous Memory Interface Control Registers 5-57
Asynchronous Memory Global Control Register (EBIU_AMGCTL) 5-57

Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1) 5-59
Avoiding Bus Contention 5-62
ARDY Input Control 5-63
Memory Bank Select Control Register (EBIU_MBSCTL) .. 5-63
Flash Memory Bank Control Registers (EBIU_FCTL, EBIU_MODE) 5-64
Booting From Page Mode or Synchronous Flash 5-65
Access Mode Selection 5-65
Memory Mode Control (EBIU_MODE) Register 5-66
Asynchronous Flash Mode 5-66
Flash Memory Bank Control (EBIU_FCTL) Register 5-67
Asynchronous Page Mode 5-67
Synchronous Burst Mode 5-67
EBIU Arbitration Status Register (EBIU_ARBSTAT) 5-69
Programmable Timing Characteristics 5-70
Asynchronous Accesses by Core Instructions 5-70
Asynchronous Reads 5-70
Asynchronous Writes 5-72
Asynchronous Writes Followed by Reads 5-75
Adding Additional Wait States 5-77
Asynchronous Flash Mode Writes and Reads 5-78
Asynchronous Page Mode Reads 5-79
Synchronous Burst Mode Read 5-80
Bus Request and Grant 5-81

SYSTEM INTERRUPTS

Overview 6-1
Features 6-2
Interfaces 6-2
Description of Operation 6-6
Events and Sequencing 6-6
System Peripheral Interrupts 6-10
Programming Model 6-22
System Interrupt Initialization 6-22
System Interrupt Processing Summary 6-22
System Interrupt Controller Registers 6-24
System Interrupt Assignment (SIC_IARx) Registers 6-25
System Interrupt Mask (SIC_IMASKx) Registers 6-32
System Interrupt Status (SIC_ISRx) Registers 6-35
System Interrupt Wakeup (SIC_IWRx) Registers 6-37
Programming Examples 6-40
Clearing Interrupt Requests 6-41

DIRECT MEMORY ACCESS

Overview and Features 7-2
DMA Controller Overview 7-6
External Interfaces 7-8
Internal Interfaces 7-8

Peripheral DMA 7-10
Memory DMA 7-13
Handshaked Memory DMA Mode 7-16
Modes of Operation 7-16
Register-Based DMA Operation 7-17
Stop Mode 7-18
Autobuffer Mode 7-18
Two-Dimensional DMA Operation 7-19
Examples of Two-Dimensional DMA 7-20
Descriptor-Based DMA Operation 7-21
Descriptor List Mode 7-22
Descriptor Array Mode 7-22
Variable Descriptor Size 7-23
Mixing Flow Modes 7-24
Functional Description 7-25
DMA Operation Flow 7-25
DMA Startup 7-25
DMA Refresh 7-30
Work Unit Transitions 7-32
DMA Transmit and MDMA Source 7-33
DMA Receive 7-35
Stopping DMA Transfers 7-36
DMA Errors (Aborts) 7-37
DMA Control Commands 7-39

Restrictions 7-43
Transmit Restart or Finish 7-44
Receive Restart or Finish 7-44
Handshaked Memory DMA Operation 7-45
Pipelining DMA Requests 7-47
HMDMA Interrupts 7-49
DMA Performance 7-50
DMA Throughput 7-51
Memory DMA Timing Details 7-54
Static Channel Prioritization 7-54
Temporary DMA Urgency 7-54
Memory DMA Priority and Scheduling 7-56
Traffic Control 7-58
Programming Model 7-60
Synchronization of Software and DMA 7-61
Single-Buffer DMA Transfers 7-63
Continuous Transfers Using Autobuffering 7-64
Descriptor Structures 7-65
Descriptor Queue Management 7-67
Descriptor Queue Using Interrupts on Every Descriptor 7-67
Descriptor Queue Using Minimal Interrupts 7-69
Software-Triggered Descriptor Fetches 7-71

DMA Registers 7-73
DMA Channel Registers 7-73
Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers 7-77
DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers 7-79
Interrupt Status (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS) Registers 7-84
Start Address (DMAx_START_ADDR and MDMA_yy_START_ADDR) Registers 7-88
Current Address (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR) Registers 7-90
Inner Loop Count (DMAx_X_COUNT and MDMA_yy_X_COUNT) Registers 7-92
Current Inner Loop Count (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT) Registers 7-94
Inner Loop Address Increment (DMAx_X_MODIFY and MDMA_yy_X_MODIFY) Registers 7-97
Outer Loop Count (DMAx_Y_COUNT and MDMA_yy_Y_COUNT) Registers 7-99
Current Outer Loop Count (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT) Registers 7-101
Outer Loop Address Increment (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY) Registers 7-103
Next Descriptor Pointer (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR) Registers 7-106
Current Descriptor Pointer (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR) Registers 7-108

Handshake MDMA (HMDMA) Registers 7-111
Handshake MDMA Control (HMDMAx_CONTROL) Registers 7-111
Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers 7-114
Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers 7-115
Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers 7-116
Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers 7-117
Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers 7-117
Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers 7-118
DMA Traffic Control Registers 7-118
DMA Traffic Control Counter Period (DMACx_TCPER) Registers 7-119
DMA Traffic Control Counter (DMACx_TCCNT) Registers 7-119
DMA Controller 1 Peripheral Multiplexer (DMAC1_PERIMUX) Register 7-121
Programming Examples 7-122
Register-Based 2D Memory DMA 7-122
Initializing Descriptors in Memory 7-126
Software-Triggered Descriptor Fetch Example 7-129
Handshake Memory DMA Example 7-132

HOST DMA PORT

Overview 8-1
Features 8-2
Interface Overview 8-3
Description of Operation 8-3
Architecture 8-4
Functional Description 8-5
HOSTDP Configuration 8-5
HOSTDP Transactions 8-7
Host Read Status 8-8
Host Read Data and Host Write Data Operations 8-8
HOSTDP Modes of Operation 8-10
Acknowledge Mode 8-10
Interrupt Mode 8-14
DMA STOP Mode and AUTOBUFFER Mode 8-15
Bus Widths and Endian Order 8-16
Access Control 8-17
Improving HOSTDP DMA Bus Bandwidth 8-18
Control Commands Between the External Host and HOSTDP 8-20
Programming Model 8-21
ADSP-BF54x processor Slave 8-21
Host Processor 8-22

Host DMA Port Registers 8-24
Host DMA Port Control (HOST_CONTROL) Register 8-25
Host DMA Port Status (HOST_STATUS) Register 8-27
HOSTDP Timeout (HOST_TIMEOUT) Register 8-29
Programming Examples 8-30

GENERAL-PURPOSE PORTS

Overview 9-1
Features 9-2
Module Overview 9-3
External Interfaces 9-4
Internal Interfaces 9-4
Pin Multiplexing Scheme 9-4
Port A 9-9
Port B 9-10
Port C 9-12
Port D 9-13
Port E 9-14
Port F 9-15
Port G 9-17
Port H 9-18
Port I 9-20
Port J 9-21
Port Multiplexing Control 9-22

GPIO Functionality 9-24
Input Mode 9-24
Output Mode 9-25
Open-Drain Mode 9-25
Pin Interrupts 9-26
Programming Model 9-29
Port Registers 9-33
Port Multiplexing Registers 9-45
Port x Function Enable (PORTx_FER) Registers 9-45
Port Multiplexer Control (PORTx_MUX) Registers 9-46
GPIO Registers 9-48
Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers 9-48
Port x GPIO Input Enable (PORTx_INEN) Registers 9-50
Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers 9-51
Pin Interrupt Registers 9-53
Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs 9-54
Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers 9-55
Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs 9-58
Pin Interrupt Pin State (PINTx_PINSTATE) Register 9-60

Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers 9-61
Pin Interrupt Assignment (PINTx_ASSIGN) Registers 9-63
Programming Examples 9-66

GENERAL-PURPOSE TIMERS

Overview and Features 10-1
Features 10-2
Interface Overview 10-3
External Interface 10-4
Internal Interface 10-5
Description of Operation 10-6
Interrupt Processing 10-7
Illegal States 10-10
Modes of Operation 10-13
Pulse Width Modulation (PWM_OUT) Mode 10-13
Output Pad Disable 10-15
Single Pulse Generation 10-15
Pulse-Width Modulation Waveform Generation 10-16
PULSE_HI Toggle Mode 10-18
Externally-Clocked PWM_OUT 10-22
Stopping the Timer in PWM_OUT Mode 10-23
Pulse-Width Count and Capture (WDTH_CAP) Mode 10-25
Autobaud Mode 10-33
Capturing Timings from the GP Counter Module 10-34

External Event (EXT_CLK) Mode 10-34
Programming Model 10-35
Timer Registers 10-37
Timer Enable (TIMER_ENABLEx) Registers 10-38
Timer Disable (TIMER_DISABLEx) Registers 10-39
Timer Status (TIMER_STATUSx) Registers 10-40
Timer Configuration (TIMERx_CONFIG) Registers 10-42
Timer Counter (TIMERx_COUNTER) Registers 10-44
TIMERx_PERIOD and TIMERx_WIDTH Registers 10-47
Summary 10-51
Programming Examples 10-53

CORE TIMER

Overview and Features 11-1
Timer Overview 11-2
External Interfaces 11-2
Internal Interfaces 11-2
Description of Operation 11-3
Interrupt Processing 11-3
Core Timer Registers 11-4
Core Timer Control (TCNTL) Register 11-5
Core Timer Count (TCOUNT) Register 11-5
Core Timer Period (TPERIOD) Register 11-6
Core Timer Scale (TSCALE) Register 11-7
Programming Examples 11-7

WATCHDOG TIMER

Overview and Features 12-1
Interface Overview 12-3
External Interface 12-3
Internal Interface 12-3
Description of Operation 12-4
Watchdog Timer Registers 12-6
Watchdog Count (WDOG_CNT) Register 12-6
Watchdog Status (WDOG_STAT) Register 12-7
Watchdog Control (WDOG_CTL) Register 12-8
Programming Examples 12-9

ROTARY COUNTER

Overview 13-1
Features 13-2
Interface Overview 13-3
Description of Operation 13-4
Quadrature Encoder Mode 13-4
Binary Encoder Mode 13-5
Rotary Counter Mode 13-6
Direction Counter Mode 13-7
Timed Direction Mode 13-7

Functional Description 13-8
Input Noise Filtering (Debouncing) 13-8
Zero Marker (Pushbutton) Operation 13-12
Boundary Comparison Modes 13-13
Rotary Encoder Events: Control and Signaling 13-15
Illegal Gray/Binary Code Events (Two-Step Detection) 13-16
Up/Down Count Events 13-16
Zero Count Events 13-17
Overflow Events 13-17
Boundary Match Events 13-17
Zero Marker Events 13-18
Capturing Timing Information (Using the General-Purpose Timer) 13-18
Capturing Time Interval Between Successive Counter Events 13-19
Capturing Counter Interval and CNT_COUNTER Read Timing 13-20
Counter Commands 13-23
Programming Mode 13-24
Rotary Counter Registers 13-24
Configuration (CNT_CONFIG) Register 13-26
Boundary Register Mode 13-26
Interrupt Mask (CNT_IMASK) Register 13-28
Status (CNT_STATUS) Register 13-28
Command (CNT_COMMAND) Register 13-29

Debounce Prescale (CNT_DEBOUNCE) Register 13-30
Counter (CNT_COUNTER) Register 13-31
Boundary (CNT_MIN and CNT_MAX) Registers 13-32
Programming Examples 13-33

REAL-TIME CLOCK

Overview 14-1
Interface Overview 14-3
Description of Operation 14-3
RTC Clock Requirements 14-3
Prescaler Enable 14-4
RTC Programming Model 14-6
Register Writes 14-8
Write Latency 14-9
Register Reads 14-10
Deep Sleep 14-10
Event Flags 14-11
Setting Time of Day 14-13
Using the Stopwatch 14-14
Interrupts 14-15
State Transitions Summary 14-17
RTC Registers 14-20
RTC Status (RTC_STAT) Register 14-21
RTC Interrupt Control (RTC_ICTL) Register 14-21
RTC Interrupt Status (RTC_ISTAT) Register 14-22

RTC Stopwatch Count (RTC_SWCNT) Register 14-22
RTC Alarm (RTC_ALARM) Register 14-23
RTC Prescaler Enable (RTC_PREN) Register 14-23
Programming Examples 14-24
Enable RTC Prescaler 14-24
RTC Stopwatch For Exiting Deep Sleep Mode 14-25
RTC Alarm to Come Out of Hibernate State 14-27

ENHANCED PARALLEL PERIPHERAL INTERFACE

Overview 15-1
Interface Overview 15-4
Description of Operation 15-6
EPPI Reset 15-7
Clock Gating 15-8
Frame Sync Polarity & Sampling Edge 15-8
Interrupts 15-9
Functional Description 15-10
ITU-R 656 Modes 15-10
ITU-R 656 Background 15-10
ITU-R 656 Input Modes 15-15
Entire Field 15-16
Active Video 15-16
Vertical Blanking Interval (VBI) only 15-17
ITU-R 656 Output in GP Transmit Modes 15-18
Frame Synchronization in ITU-R 656 Modes 15-21

General-Purpose EPPI Modes 15-22
GP 0 FS Mode 15-22
Frame Synchronization in GP 0 FS External Trigger Mode 15-23
Frame Synchronization in GP 0 FS Internal Trigger Mode 15-23
GP 1 FS Mode 15-24
GP 2 FS Mode 15-24
DEN functionality in GP 2 FS Transmit Mode 15-26
GP 3 FS Mode 15-27
EPPI Data Path Options 15-27
EPPI Data Lengths 15-27
EPPI DMA Channels 15-28
Data Packing For Receive Modes 15-28
Data Unpacking For Transmit Modes 15-29
Sign-Extension and Zero-Filling 15-30
Split Receive Modes 15-31
Split Transmit Modes 15-31
RGB Data Formats 15-32
Programmed Clipping and Thresholding of Data Values 15-32
Data Transfer Examples 15-33
8-Bit Receive Mode 15-33
10/12/14-Bit Receive Modes 15-35
16-Bit Receive Mode 15-38
18-Bit Receive Mode 15-40

24-Bit Receive Mode 15-42
8-Bit Split Receive Mode 15-43
10/12/14/16-Bit Split Receive Mode with SPLT_16 = 0 .. 15-47
16-Bit Split Receive Mode with SPLT_16 = 1 15-49
8-Bit Transmit Mode 15-50
10/12/14-Bit Transmit Modes 15-51
16-Bit Transmit Mode 15-52
18-Bit Transmit Mode 15-53
24-Bit Transmit Mode 15-54
8-Bit Split Transmit Mode 15-55
10/12/14/16-Bit Split Transmit Mode with SPLT_16 = 0 15-59
16-Bit Split Transmit Mode with SPLT_16 = 1 15-63
Programming Model 15-64
DMA Operation 15-64
Elevating EPPI Urgent Requests at DDR Controller Interface 15-71
System Configuration 15-73
EPPI Registers 15-73
EPPI Status (EPPIx_STATUS) Register 15-77
EPPIx Control (EPPIx_CONTROL) Register 15-80
Windowing Registers 15-88
EPPI Lines per Frame Register (EPPIx_FRAME) 15-90
EPPI Samples per Line Register (EPPIx_LINE) 15-90
EPPI Vertical Delay Register (EPPIx_VDELAY) 15-91

EPPI Vertical Transfer Count Register (EPPIx_VCOUNT) 15-91
EPPI Horizontal Delay Register (EPPIx_HDELAY) 15-92
EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT) 15-93
EPPI Clock Divide Register (EPPIx_CLKDIV) 15-93
Frame Sync/ Blanking Generation Registers 15-94
EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL) 15-94
EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB) 15-95
EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL) 15-96
EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF) 15-97
EPPI Clipping Register (EPPIx_CLIP) 15-98

SECURITY

Overview 16-1
Features 16-4
Description of Operation 16-6
Secure State Machine 16-7
Open Mode 16-8
Secure Entry Mode 16-8
Secure Mode 16-9
SecureMode Control 16-10

Security Features 16-12
Digital Signature Authentication 16-13
Digital Signature Authentication Performance Measurement 16-16
Protection Features 16-17
Operating in Secure Mode 16-20
Entering Secure Mode 16-20
Exiting Secure Mode 16-20
Reset Handling in Secure Mode 16-21
Hardware Reset 16-21
Clearing Private Data 16-22
Public Key Requirements 16-23
Storing Public Cipher Key in Public OTP 16-25
Cryptographic Ciphers 16-26
Keys 16-26
Debug Functionality 16-27
Programming Examples 16-31
Programming Model 16-32
Secure Entry Service Routine (SESR) API 16-32
Starting Authentication 16-32
Memory Configuration 16-35
Message Placement 16-35
Digital Signature 16-36
Message Size Constraints 16-36

Memory Usage 16-37
Memory Protection 16-37
Secure Function and Secure Entry Service Routine Arguments 16-38
Secure Function Arguments 16-38
Secure Entry Service Routine Arguments 16-39
usFlags 16-39
uslRQMask 16-40
ulMessageSize 16-41
ulSFEntryPoint 16-41
ulMessagePtr 16-41
Secure Message Execution 16-41
Return Codes 16-42
Advanced Encryption Standard (AES) API 16-44
ADI_AES_DATA Data Type 16-44
ADI_AES_KEYEXPANSION Data Type 16-46
ADI_AES_CIPHER Data Type 16-47
bfrom_AesInit() ROM Routine 16-48
bfrom_AesKeyexp() ROM Routine 16-49
bfrom_AesInvKeyexp() ROM Routine 16-50
bfrom_AesCipher() ROM Routine 16-50
bfrom_AesInvCipher() ROM Routine 16-51
SECURE HASH ALGORITHM (SHA-1) API 16-52
ADI_SHA1 Data Type 16-52

bfrom_Sha1Init ROM Routine 16-53
bfrom_Sha1Hash ROM Routine 16-53
ARC4 API 16-53
ADI_ARC4_KEY Data Type 16-54
ADI_ARC4_DATA Data Type 16-54
bfrom_Arc4Init ROM Routine 16-55
bfrom_Arc4Cipher ROM Routine 16-55
Security Registers 16-56
Secured System Switches (SECURE_SYSSWT) Register 16-57
Secure Control (SECURE_CONTROL) Register 16-64
Secure Status (SECURE_STATUS) Register 16-67

SYSTEM RESET AND BOOTING

Overview 17-1
Reset and Power-up 17-4
Hardware Reset 17-6
Software Resets 17-7
Reset Vector 17-8
Servicing Reset Interrupts 17-10
Preboot 17-11
Factory Page Settings (FPS) 17-14
Preboot Page Settings (PBS) 17-14
Alternative PBS Pages 17-16
Programming PBS Pages 17-16
Recovering From Misprogrammed PBS Pages 17-17

Customizing Power Management 17-17
Customizing Booting Options 17-18
Customizing the Asynchronous Port 17-20
Customizing the Synchronous Port 17-21
Basic Booting Process 17-23
Block Headers 17-25
Block Code 17-27
DMA Code Field 17-27
Block Flags Field 17-29
Header Checksum Field 17-30
Header Sign Field 17-31
Target Address 17-31
Byte Count 17-32
Argument 17-33
Boot Host Wait (HWAIT) Feedback Strobe 17-33
Using HWAIT as Reset Indicator 17-35
Boot Termination 17-35
Single Block Boot Streams 17-36
Direct Code Execution 17-37
Advanced Boot Techniques 17-39
Initialization Code 17-39
Quick Boot 17-43
Indirect Booting 17-44
Callback Routines 17-45

Error Handler 17-48
CRC Checksum Calculation 17-48
Load Functions 17-49
Calling the Boot Kernel at Runtime 17-50
Debugging the Boot Process 17-51
Boot Management 17-54
Booting a Different Application 17-54
Multi-DXE Boot Streams 17-56
Determining Boot Stream Start Addresses 17-60
Initialization Hook Routine 17-60
Specific Boot Modes 17-61
No Boot Mode 17-62
Flash Boot Modes 17-62
SDRAM Boot Mode 17-66
FIFO Boot Mode 17-67
SPI Master Boot Modes 17-69
SPI Device Detection Routine 17-71
SPI Slave Boot Mode 17-73
TWI Master Boot Mode 17-77
TWI Slave Boot Mode 17-79
UART Slave Mode Boot 17-82
OTP Boot Mode 17-85
Host DMA Boot Modes 17-86

NAND Flash Boot Mode 17-88
Supported Devices 17-91
NAND Flash Page Structure 17-94
Auto Detection 17-95
Boot Stream Processing 17-96
Software Configurable NAND Flash Boot Modes 17-98
Sequential Block Mode 17-98
Block Skip Mode 17-99
Multiple Image Mode 17-101
Reset and Booting Registers 17-103
Software Reset (SWRST) Register 17-103
System Reset Configuration (SYSCR) Register 17-105
Boot Code Revision Control (BK_REVISION) 17-107
Boot Code Date Code (BK_DATECODE) 17-108
Zero Word (BK_ZEROS) 17-109
Ones Word (BK_ONES) 17-110
OTP Memory Pages for Booting 17-110
Lower PBS00 Half Page 17-110
Upper PBS00 Half Page 17-114
Lower PBS01 Half Page 17-115
Upper PBS01 Half Page 17-116
Lower PBS02 Half Page 17-118
Upper PBS02 Half Page 17-119
Reserved Half Pages 17-120

Data Structures 17-120
ADI_BOOT_HEADER 17-120
ADI_BOOT_BUFFER 17-121
ADI_BOOT_DATA 17-121
dFlags Word 17-125
ADI_BOOT_NAND 17-126
ADI_BOOT_NAND_DEVICE 17-128
ADI_BOOT_NAND_BUFFER 17-130
ADI_BOOT_NAND_ACCESS 17-131
ADI_BOOT_NAND_ADDRESS 17-132
ADI_BOOT_NAND_ECC 17-134
Callable ROM Functions for Booting 17-136
BFROM_FINALINIT 17-136
BFROM_PDMA 17-136
BFROM_MDMA 17-136
BFROM_MEMBOOT 17-137
BFROM_TWIBOOT 17-138
BFROM_SPIBOOT 17-139
BFROM_OTPBOOT 17-140
BFROM_NANDBOOT 17-141
BFROM_BOOTKERNEL 17-142
BFROM_CRC32 17-142
BFROM_CRC32POLY 17-143

BFROM_CRC32CALLBACK 17-144
BFROM_CRC32INITCODE 17-144
Programming Examples 17-145
System Reset 17-145
Exiting Reset to User Mode 17-146
Exiting Reset to Supervisor Mode 17-146
Initcode (SDRAM Controller Setup) 17-147
Initcode (Power Management Control) 17-149
Initcode (NAND Flash Boot Mode Configuration) 17-150
Quickboot With Restore From SDRAM 17-151
XOR Checksum 17-152
Direct Code Execution 17-154
Managing PBS Pages in OTP Memory 17-155

DYNAMIC POWER MANAGEMENT

Phase-Locked Loop and Clock Control 18-1
PLL Overview 18-2
PLL Clock Multiplier Ratios 18-3
Core Clock/System Clock Ratio Control 18-4
Dynamic Power Management Controller 18-7
Operating Modes 18-7
Dynamic Power Management Controller States 18-8
Full On Mode 18-8
Active Mode 18-9
Sleep Mode 18-9

Deep Sleep Mode 18-10
Hibernate State 18-11
Operating Mode Transitions 18-11
Programming Operating Mode Transitions 18-14
Dynamic Supply Voltage Control 18-16
Power Supply Management 18-16
Controlling the Voltage Regulator 18-17
Changing Voltage 18-19
Powering Down the Core (Hibernate State) 18-20
Recovery From Hibernate State 18-23
PLL and VR Registers 18-25
PLL Divide (PLL_DIV) Register 18-26
PLL Control (PLL_CTL) Register 18-26
PLL Status (PLL_STAT) Register 18-27
PLL Lock Count (PLL_LOCKCNT) Register 18-27
Voltage Regulator Control (VR_CTL) Register 18-28
System Control ROM Function 18-29
Programming Model 18-31
Access System Control ROM Function in C/C++ 18-31
Access System Control ROM Function in Assembly 18-32
Programming Examples 18-35
Full On Mode to Active Mode and Back 18-36
Transition to Sleep Mode or Deep Sleep Mode 18-38
Setting Wakeups and Entering Hibernate State 18-40

Perform a System Reset or Soft-Reset 18-43
Change VCO, Core Clock, and System Clock Frequency 18-44
Changing Voltage Levels 18-46

SYSTEM DESIGN

Pin Descriptions 19-1
Managing Clocks 19-2
Managing Core and System Clocks 19-2
Configuring and Servicing Interrupts 19-2
Semaphores 19-3
Example Code for Query Semaphore 19-4
Data Delays, Latencies, and Throughput 19-4
Bus Priorities 19-5
System-Level Hardware Design 19-5
External Memory Design Issues 19-5
DDR Memory 19-5
Memory Bus Pin Muxing and Flow Control 19-7
Example Asynchronous Memory Interfaces 19-8
Avoiding Bus Contention 19-9
BURST FLASH 19-10
NAND FLASH 19-11
USB Controller 19-11
ATAPI Bus 19-13
Voltage Regulator 19-13

Signal Integrity 19-14
Decoupling Capacitors and Ground Planes 19-15
5 Volt Tolerance 19-17
Resetting the Processor 19-17
Recommendations for Unused Pins 19-17
Programmable Outputs and Pin Multiplexing 19-17
Test Point Access 19-18
Oscilloscope Probes 19-18
Recommended Reading 19-19

NAND FLASH CONTROLLER

Overview 20-2
Interface Overview 20-4
Description of Operation 20-5
Internal Bus Interfaces 20-5
Bus Access Types 20-6
Access Timing 20-6
Pin Sharing 20-7
Functional Description 20-8
Page Write 20-8
Page Read 20-9
Additional Operations 20-10
Write Protection 20-11
Chip Enable Don't Care 20-11

NFC Error Detection 20-11
Error Analysis 20-13
Large Page Size Support 20-15
NFC SmartMedia Support 20-15
Programming Model 20-15
NFC Registers 20-17
NFC Control Register (NFC_CTL) 20-19
NFC Status Register (NFC_STAT) 20-20
NFC Interrupt Status Register (NFC_IRQSTAT) 20-21
NFC Interrupt Mask Register (NFC_IRQMASK) 20-22
NFC ECC Registers (NFC_ECCx) 20-22
NFC Count Register (NFC_COUNT) 20-24
NFC Reset Register (NFC_RST) 20-24
NFC Page Control Register (NFC_PGCTL) 20-25
NFC Read Data Register (NFC_READ) 20-25
NFC Address Register (NFC_ADDR) 20-26
NFC Command Register (NFC_CMD) 20-27
NFC Data Write Register (NFC_DATA_WR) 20-28
NFC Data Read Register (NFC_DATA_RD) 20-28
NFC Programming Examples 20-29

ATAPI INTERFACE

Interface Overview 21-1

Description of Operation 21-4
Host PIO/Register Transfers 21-4
PIO Data-Out Transfers (Device Write) 21-5
PIO Data-In Transfers (Device Read) 21-8
Host Multiword DMA Transfers 21-10
Host Pausing the Multi-DMA Transfer 21-13
Host Terminating the Multi DMA Transfer 21-13
Device Pausing the Multi-DMA Transfer 21-13
Device Terminating the Multi-DMA Transfer 21-14
Host Ultra DMA Command Protocol Transfers 21-15
Host Pausing the Ultra DMA Data-In Transfer 21-16
Host Terminating the Ultra DMA Data-In Transfer 21-16
Device Pausing the Ultra DMA Data-In Transfer 21-16
Device Terminating the Ultra DMA Data-In Transfer 21-17
Host Pausing Ultra DMA Data-Out Transfer 21-17
Host Terminating Ultra DMA Data-Out Transfer 21-17
Device Pausing the Ultra DMA Data-Out Transfer 21-17
Device Terminating the Ultra DMA Data-Out Transfer ... 21-18
Functional Description 21-18
Power-on and Hardware Reset Protocol 21-18
Device Selection Protocol 21-19
Programmed I/O (PIO) 21-21
Host Multi DMA Block Implementation 21-22

Host Ultra DMA Block Implementation 21-27
Initiating an Ultra DMA Data-In Burst 21-27
Data-In Transfer 21-30
Device pausing an Ultra DMA Data-In Burst 21-31
Host pausing an Ultra DMA Data-In Burst 21-31
Ultra DMA Timing 21-32
Ultra DMA-Out Timing 21-37
Programming Model 21-40
ATAPI Device Configuration and Setup 21-40
PIO Data-out Transfers Pseudo-code 21-43
Host Multiword DMA Transfers Pseudo-code 21-44
Host Ultra DMA Command Protocol Transfers Pseudo-code 21-45
ATAPI Registers 21-46
ATAPI Control and Status Registers 21-49
ATAPI Control (ATAPI_CONTROL) Register 21-49
ATAPI Status (ATAPI_STATUS) Register 21-51
ATAPI Device Address (ATAPI_DEV_ADDR) Register ... 21-52
ATAPI Device Transmit Buffer (ATAPI_DEV_TXBUF) Register 21-53
ATAPI Device Receive Buffer (ATAPI_DEV_RXBUF) Register 21-54
ATAPI Interrupt Mask (ATAPI_INT_MASK) Register 21-54
ATAPI Interrupt Status (ATAPI_INT_STATUS) Register 21-56

ATAPI Transfer Length (ATAPI_XFER_LEN) Register 21-58
ATAPI Line Status (ATAPI_LINE_STATUS) Register 21-59
ATAPI State Machine Status (ATAPI_SM_STATE) Register 21-59
ATAPI Host Terminate (ATAPI_TERMINATE) Register 21-60
ATAPI PIO Transfer Count (ATAPI_PIO_TFRCNT) Register 21-61
ATAPI Multiword DMA Transfer Count (ATAPI_MULTI_TFRCNT) Register 21-61
ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT) Register 21-62
ATAPI Ultra DMA OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT) Register 21-63
ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0) Register 21-63
ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0) Register 21-64
ATAPI Programmed I/O Timing 1 (ATAPI_PIO_TIM_1) Register 21-64
ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0) Register 21-65
ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1) Register 21-65
ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2) Register 21-66
ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0) Register 21-66

ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1) Register 21-67
ATAPI Ultra DMA Timing 2 (ATAPI_ULTRA_TIM_2) Register 21-67
ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3) Register 21-68
ATAPI Device I/O Registers 21-68
Command Register (R/W) 21-70
Device Control Register (WO) 21-71
Features Register (WO) 21-71
Sector Count Register (R/W) 21-71
Status Register (RO) 21-72
Alternate Status Register (RO) 21-73
Error Register (RO) 21-73
ATAPI Standards Reference 21-73
Summary of IDE/ATA Standards 21-77
ATAPI Timing Summary 21-78
IDE/ATA Transfer Modes and Protocols 21-78
Programmed (I/O) PIO Modes 21-78
Direct Memory Access (DMA) Modes 21-79
Ultra Direct Memory Access (DMA) Modes 21-79
ATAPI Device Selection 21-80

SPI-COMPATIBLE PORT CONTROLLERS

Overview 22-1

Interface Overview 22-3
External Interface 22-4
Serial Peripheral Interface Clock Signal (SPIxSCK) 22-5
Master Out Slave In (MOSI) 22-6
Master In Slave Out (MISO) 22-6
Serial Peripheral Interface Slave Select Input Signal 22-7
Serial Peripheral Interface Slave Select Enable Output Signals 22-8
Slave Select Inputs 22-12
Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems 22-12
Internal Interfaces 22-14
DMA Functionality 22-14
SPI Transmit Data Buffer 22-15
SPI Receive Data Buffer 22-16
Description of Operation 22-16
SPI Transfer Protocols 22-17
SPI General Operation 22-19
SPI Control 22-21
Clock Signals 22-22
SPI Baud Rate 22-22
Error Signals and Flags 22-23
Mode Fault Error (MODF) 22-24
Transmission Error (TXE) 22-25

Reception Error (RBSY) 22-25
Transmit Collision Error (TXCOL) 22-25
Interrupt Output 22-25
Functional Description 22-26
Master Mode Operation 22-26
Transfer Initiation From Master (Transfer Modes) 22-28
Slave Mode Operation 22-29
Slave Ready for a Transfer 22-30
Programming Model 22-30
Beginning and Ending an SPI Transfer 22-31
Master Mode DMA Operation 22-33
Slave Mode DMA Operation 22-35
SPI Registers 22-43
SPI Baud Rate (SPIx_BAUD) Register 22-44
SPI Control (SPIx_CTL) Register 22-45
SPI Flag (SPIx_FLG) Register 22-46
SPI Status (SPIx_STAT) Register 22-48
SPI Transmit Data Buffer (SPIx_TDBR) Register 22-48
SPI Receive Data Buffer (SPIx_RDBR) Register 22-49
SPI RDBR Shadow (SPIx_SHADOW) Register 22-49
Programming Examples 22-49
Core Generated Transfer 22-50
Initialization Sequence 22-50
Starting a Transfer 22-51

Post Transfer and Next Transfer 22-52
Stopping 22-53
DMA Transfer 22-53
DMA Initialization Sequence 22-53
SPI Initialization Sequence 22-54
Starting a Transfer 22-56
Stopping a Transfer 22-56

TWO-WIRE INTERFACE CONTROLLERS

Overview 23-2
Interface Overview 23-3
External Interface 23-4
Serial Clock signal (SCL1–0) 23-4
Serial data signal (SDA1–0) 23-4
TWI Pins 23-5
Internal Interfaces 23-5
Description of Operation 23-6
TWI Transfer Protocols 23-6
Clock Generation and Synchronization 23-7
Bus Arbitration 23-8
Start and Stop Conditions 23-9
General Call Support 23-10
Fast Mode 23-10

TWI General Operation 23-11
TWI Control 23-11
Clock Signal 23-12
Functional Description 23-13
General Setup 23-13
Slave Mode 23-13
Master Mode Clock Setup 23-14
Master Mode Transmit 23-15
Master Mode Receive 23-16
Clock Stretching 23-17
Clock Stretching During FIFO Underflow 23-18
Clock Stretching during FIFO Overflow 23-19
Clock Stretching During Repeated Start Condition 23-21
Programming Model 23-23
TWI Registers 23-25
SCLx Clock Divider (TWIx_CLKDIV) Register 23-26
TWI Control (TWIx_CONTROL) Register 23-27
TWI Slave Mode Control (TWIx_SLAVE_CTL) Register 23-27
TWI Slave Mode Address (TWIx_SLAVE_ADDR) Register 23-30
TWI Slave Mode Status (TWIx_SLAVE_STAT) Register 23-30
TWI Master Mode Control (TWIx_MASTER_CTL) Register 23-32
TWI Master Mode Address (TWIx_MASTER_ADDR) Register 23-35

TWI Master Mode Status (TWIx_MASTER_STAT) Register 23-35
TWI FIFO Control (TWIx_FIFO_CTL) Register 23-39
TWI FIFO Status (TWIx_FIFO_STAT) Register 23-41
TWI FIFO Status 23-41
TWI Interrupt Mask (TWIx_INT_MASK) Register 23-43
TWI Interrupt Status (TWIx_INT_STAT) Register 23-44
TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Register 23-48
TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Register 23-49
TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Register 23-50
TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Register 23-51
Programming Examples 23-52
Master Mode Setup 23-52
Slave Mode Setup 23-57
Electrical Specifications 23-63

SPORT CONTROLLERS

Overview 24-1
Features 24-2
Interface Overview 24-4
SPORT Pin/Line Terminations 24-10

Description of Operation 24-11
SPORT Operation 24-11
SPORT Disable 24-11
Setting SPORT Modes 24-12
Stereo Serial Operation 24-13
Multichannel Operation 24-17
Multichannel Enable 24-19
Frame Syncs in Multichannel Mode 24-20
Multichannel Frame 24-22
Multichannel Frame Delay 24-23
Window Size 24-23
Window Offset 24-24
Other Multichannel Fields in SPORTx_MCMC2 24-24
Channel Selection Register 24-25
Multichannel DMA Data Packing 24-26
Support for H.100 Standard Protocol 24-27
2X Clock Recovery Control 24-27
Functional Description 24-28
Clock and Frame Sync Frequencies 24-28
Maximum Clock Rate Restrictions 24-29
Word Length 24-30
Bit Order 24-30
Data Type 24-30
Companding 24-31

Clock Signal Options 24-32
Frame Sync Options 24-33
Framed Versus Unframed 24-33
Internal Versus External Frame Syncs 24-34
Active Low Versus Active High Frame Syncs 24-35
Sampling Edge for Data and Frame Syncs 24-35
Early Versus Late Frame Syncs (Normal Versus Alternate Timing) 24-38
Data Independent Transmit Frame Sync 24-40
Moving Data Between SPORTs and Memory 24-40
SPORT RX, TX, and Error Interrupts 24-41
PAB Errors 24-41
Timing Examples 24-42
SPORT Registers 24-48
Register Writes and Effective Latency 24-50
Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers 24-51
SPORTx_RCR1 and SPORTx_RCR2 Registers 24-56
Data Word Formats 24-61
Transmit Data (SPORTx_TX) Register 24-61
Receive Data (SPORTx_RX) Register 24-64
SPORT Status (SPORTx_STAT) Register 24-66
Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers 24-68

Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers 24-69
Multichannel Configuration (SPORTx_MCMCn) Registers 24-70
Current Channel (SPORTx_CHNL) Register 24-71
Multichannel Selection Receive (SPORTx_MRCSn) Registers 24-72
Multichannel Selection Transmit (SPORTx_MTCSn) Registers 24-74
Programming Examples 24-76
SPORT Initialization Sequence 24-77
DMA Initialization Sequence 24-79
Interrupt Servicing 24-81
Starting a Transfer 24-82

UART PORT CONTROLLERS

Overview 25-1
Features 25-2
Interface Overview 25-3
External Interface 25-3
Internal Interface 25-5
Description of Operation 25-6
UART Transfer Protocol 25-6
UART Transmit Operation 25-7
UART Receive Operation 25-8
Hardware Flow Control 25-10

IrDA Transmit Operation 25-13
IrDA Receive Operation 25-14
Interrupt Processing 25-15
Bit Rate Generation 25-18
Autobaud Detection 25-20
Programming Model 25-22
Non-DMA Mode 25-22
DMA Mode 25-24
Mixing Modes 25-26
UART Registers 25-26
Line Control (UARTx_LCR) Registers 25-29
Modem Control (UARTx_MCR) Registers 25-32
Line Status (UARTx_LSR) Registers 25-34
Modem Status (UARTx_MSR) Registers 25-37
Transmit Hold (UARTx_THR) Registers 25-39
Receive Buffer (UARTx_RBR) Registers 25-40
Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers 25-40
Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers 25-46
UART Scratch (UARTx_SCR) Registers 25-49
Global Control (UARTx_GCTL) Registers 25-50
Programming Examples 25-51

USB OTG CONTROLLER

Overview 26-1
Features 26-2
Interface Overview 26-3
FIFO Configuration 26-7
Interrupts 26-8
Resets 26-11
Description of Operation 26-12
Peripheral Mode Operation 26-13
Endpoint Setup 26-13
IN Transactions as a Peripheral 26-14
OUT Transactions as a Peripheral 26-15
Peripheral Transfer Workflows 26-17
Control Transactions as a Peripheral 26-18
Write Requests 26-19
Read Requests 26-20
Zero Data Requests 26-21
ENDPOINT 0 States 26-22
Endpoint 0 Service Routine as Peripheral 26-24
Peripheral Mode, Bulk IN, Transfer Size Known 26-31
Peripheral Mode, Bulk IN, Transfer Size Unknown 26-31
Peripheral Mode, ISO IN, Small *MaxPktSize* 26-32
Peripheral Mode, ISO IN, Large *MaxPktSize* 26-33
Peripheral Mode, Bulk OUT, Transfer Size Known 26-34

Peripheral Mode, Bulk OUT, Transfer Size Unknown .. 26-34
Peripheral Mode, ISO OUT, Small *MaxPktSize* 26-35
Peripheral Mode, ISO OUT, Large *MaxPktSize* 26-36
Peripheral Mode Suspend 26-36
Start-of-frame (SOF) Packets 26-37
Soft Connect/Soft Disconnect 26-37
Error Handling As a Peripheral 26-38
Stalls Issued to Control Transfers 26-39
Zero Length OUT Data Packets in Control Transfers 26-40
Host Mode Operation 26-40
Endpoint Setup and Data Transfer 26-40
Control Transaction as a Host 26-41
Setup Phase as a Host 26-42
IN Data Phase as a Host 26-43
OUT Data as a Host (Control) 26-44
IN Status Phase as a Host (Following SETUP Phase or OUT Data Phase) 26-45
OUT Status Phase as a Host (following IN Data Phase) ... 26-46
Host IN Transactions 26-47
Host OUT Transactions 26-48
Transaction Scheduling 26-49
Babble 26-50
VBUS Events 26-51
Actions as an “A” Device 26-51
Actions as a “B” Device 26-52

Host Mode Reset 26-53
Host Mode Suspend 26-53
Functional Description 26-54
On-Chip Bus Interfaces 26-54
Interface Pins 26-55
Power and Clocking 26-55
UTMI Interface 26-56
Programming Model 26-56
Peripheral Mode Flow Charts 26-57
Host Mode Flow Charts 26-66
DMA Mode Flow Charts 26-75
OTG Session Request 26-80
Starting a Session 26-80
Detecting Activity 26-81
Host Negotiation/Configuration 26-82
Software Clock Control 26-83
Wakeup from Hibernate State 26-84
Wakeup Without Re-Enumeration 26-85
Data Transfer 26-88
Loading/Unloading Packets from Endpoints 26-88
DMA Master Channels 26-90
DMA Bus Cycles 26-92

Transferring Packets Using DMA 26-92
Individual Packet: RX Endpoint 26-93
Individual Packet: TX Endpoint 26-94
Multiple Packets: RX Endpoint 26-94
Multiple Packets: TX Endpoints 26-96
USB OTG Registers 26-97
USB Global Control (USB_GLOBAL_CTL) Register 26-97
USB Power Management (USB_POWER) Register 26-99
USB Function Address (USB_FADDR) Register 26-102
USB Test Mode (USB_TESTMODE) Register 26-103
USB Global Interrupt (USB_GLOBINTR) Register 26-104
USB Transmit Interrupt (USB_INTRTX) Register 26-105
USB Receive Interrupt (USB_INTRRX) Register 26-106
USB Transmit Interrupt Enable (USB_INTRTXE) Register 26-107
USB Receive Interrupt Enable (USB_INTRRXE) Register 26-108
USB Common Interrupts (USB_INTRUSB) Register 26-109
USB Common Interrupt Enable (USB_INTRUSBE) Register 26-110
USB Frame Number (USB_FRAME) Register 26-111
USB Index (USB_INDEX) Register 26-111
USB TX Max Packet (USB_TX_MAX_PACKET) Register 26-112
USB Control/Status EP0 (USB_CSR0) Register 26-113

USB TX Control/Status EPx (USB_TXCSR) Register 26-117
USB RX Max Packet (USB_RX_MAX_PACKET) Register 26-122
USB RX Control/Status (USB_RXCSR) Register 26-123
USB Count 0 (USB_COUNT0) Register 26-128
USB RX Byte Count EPx (USB_RXCOUNT) Register 26-128
USB TX Type (USB_TXTYPE) Register 26-129
USB NAK Limit 0 (USB_NAKLIMIT0) Register 26-130
USB TX Interval (USB_TXINTERVAL) Register 26-130
USB RX Type (USB_RXTYPE) Register 26-131
USB RX Interval (USB_RXINTERVAL) Register 26-132
USB TX Byte Count EPx (USB_TXCOUNT) Register 26-133
USB Endpoint FIFO (USB_EPx_FIFO) Registers 26-134
USB OTG Device Control (USB_OTG_DEV_CTL) Register 26-134
USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register 26-136
USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register 26-137
USB Link Info (USB_LINKINFO) Register 26-138
USB VBUS Pulse Length (USB_VPLEN) Register 26-139
USB High-Speed EOF 1 (USB_HS_EOF1) Register 26-139
USB Full-Speed EOF 1 (USB_FS_EOF1) Register 26-140
USB Low-Speed EOF 1 (USB_LS_EOF1) Register 26-140
USB APHY Control 2 (USB_APHY_CNTRL2) Register 26-141

USB PLL OSC Control (USB_PLLOSC_CTRL) Register 26-142
USB SRP Clock Divider (USB_SRP_CLKDIV) Register 26-143
USB DMA Interrupt (USB_DMA_INTERRUPT) Register 26-144
USB DMAx Control (USB_DMA_CONTROL) Registers 26-144
USB DMAx Address Low (USB_DMAxADDRLOW) Registers 26-146
USB DMAx Address High (USB_DMAxADDRHIGH) Registers 26-147
USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers 26-147
USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers 26-148
References 26-148
Glossary of USB Terms 26-148

SECURE DIGITAL HOST

Overview 27-1
Interface Overview 27-2
Description of Operation 27-5
Functional Description 27-8
SDH Clock Configuration 27-8
SDH Interface Configuration 27-9
Card Detection 27-10

SDH Power Saving Configuration 27-12
SDH Commands and Responses 27-14
IDLE State 27-19
PEND State 27-20
SEND State 27-20
WAIT State 27-21
RECEIVE State 27-21
SDH Command Path CRC 27-22
SDH Data 27-22
SDH Data Transmit Path 27-25
SDH Data Receive Path 27-27
SDH Data Path CRC 27-29
SDH Data FIFO 27-29
SDIO Interrupt and Read Wait Support 27-30
Programming Model 27-31
Card Identification 27-32
SD Card Identification Procedure 27-32
MMC Identification Procedure 27-34
Single Block Write Operations 27-35
Using Core 27-35
Using DMA 27-37
Single Block Read Operations 27-39
Using Core 27-40
Using DMA 27-41

Multiple Block Write Operations 27-43
Using Core 27-43
Using DMA 27-46
Multiple Block Read Operations 27-48
Using Core 27-48
Using DMA 27-50
SDH Registers 27-52
SDH Power Control Register (SDH_PWR_CTL) 27-55
SDH Clock Control Register (SDH_CLK_CTL) 27-55
SDH Argument Register (SDH_ARGUMENT) 27-57
SDH Command Register (SDH_COMMAND) 27-57
SDH Response Command Register (SDH_RESP_CMD) 27-58
SDH Response Registers (SDH_RESPONSEx) 27-59
SDH Data Timer Register (SDH_DATA_TIMER) 27-60
SDH Data Length Register (SDH_DATA_LGTH) 27-61
SDH Data Control Register (SDH_DATA_CTL) 27-61
SDH Data Counter Register (SDH_DATA_CNT) 27-62
SDH Status Register (SDH_STATUS) 27-63
SDH Status Clear Register (SDH_STATUS_CLR) 27-65
SDH Interrupt Mask Registers (SDH_MASKx) 27-66
SDH FIFO Counter Register (SDH_FIFO_CNT) 27-68
SDH Data FIFO Register (SDH_FIFO) 27-69
SDH Exception Status Register (SDH_E_STATUS) 27-69
SDH Exception Mask Register (SDH_E_MASK) 27-70

SDH Configuration Register (SDH_CFG) 27-71
SDH Read Wait Enable Register (SDH_RD_WAIT_EN) 27-72
SDH Identification Registers (SDH_PIDx) 27-73
Programming Examples 27-74

PIXEL COMPOSITOR

Overview 28-1
Features 28-2
Interface Overview 28-2
Description of Operation 28-4
General Description 28-4
Data Buffer Formats 28-6
Operation in YUV 4:2:2 Format 28-6
Operation in RGB888 Format 28-8
DMA Channels 28-9
Functional Description 28-9
Data Overlay 28-10
Transparency Control 28-17
Transparent Color 28-19
Color Space Conversion 28-20
Case 1 - Image and Overlay in the Same Format 28-21
Case 2 - Image and Overlay in Different Formats 28-22
Case 3 - Color Space Conversion Only 28-23
Color Space Conversion Matrix Equations 28-24
Color Space Converter Output Thresholds 28-25

YUV Conversion Modes 28-26
Upsampling 28-26
Downsampling 28-27
PIXC Actions 28-28
Recommendations 28-29
Special Usage Cases 28-29
Example 1 - Currently Defined Mode 28-29
Example 1 - Special Usage of This Mode 28-30
Example 2 - Currently Defined Mode 28-30
Example 2 - Special Usage of This Mode 28-31
Example 3 - Currently Defined Mode 28-32
Example 3 - Special Usage of This Mode 28-32
Example 4 - Currently Defined Mode 28-33
Example 4 - Special Usage of This Mode 28-33
Programming Model 28-34
PIXC Registers 28-35
PIXC Control (PIXC_CTL) Register 28-37
PIXC Pixels Per Line (PIXC_PPL) Register 28-38
PIXC Lines Per Frame (PIXC_LPF) Register 28-38
PIXC Horizontal Start (PIXC_xHSTART) Registers 28-39
PIXC Horizontal End (PIXC_xHEND) Registers 28-39
PIXC Vertical Start (PIXC_xVSTART) Registers 28-40
PIXC Vertical End (PIXC_xVEND) Registers 28-40
PIXC Transparency Value (PIXC_xTRANSP) Registers 28-41

PIXC Interrupt Status (PIXC_INTRSTAT) Register 28-41
PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register 28-42
PIXC G/U Conversion Coefficient (PIXC_GUCON) Register 28-43
PIXC B/V Conversion Coefficient (PIXC_BVCON) Register 28-44
PIXC Color Conversion Bias (PIXC_CCBIAS) Register 28-45
PIXC Transparency Color Value (PIXC_TC) Register 28-46

MEDIA TRANSCEIVER MODULE (MXVR)

Overview 29-1
Interface Signals 29-2
MXVR Memory Map 29-4
MXVR Registers 29-4
MXVR Configuration (MXVR_CONFIG) Register 29-13
MXVR State (MXVR_STATE_0, MXVR_STATE_1) Registers 29-19
MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0) 29-29
MXVR Interrupt Status_1 (MXVR_INT_STAT_1) Register 29-40
MXVR Interrupt Enable 0 (MXVR_INT_EN_0) Register 29-43
MXVR Interrupt Enable 1 (MXVR_INT_EN_1) Register 29-46

MXVR Node Position (MXVR_POSITION) Register 29-48
MXVR Maximum Node Position (MXVR_MAX_POSITION) Register 29-49
MXVR Node Frame Delay (MXVR_DELAY) Register 29-50
MXVR Maximum Node Frame Delay (MXVR_MAX_DELAY) Register 29-52
MXVR Logical Address (MXVR_LADDR) Register 29-53
MXVR Group Address (MXVR_GADDR) Register 29-54
MXVR Alternate Address (MXVR_AADDR) Register 29-55
MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers 29-55
MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers 29-57
MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers 29-59
MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers 29-69
MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers 29-71
MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers 29-72
MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers 29-74

MXVR Asynchronous Packet Control (MXVR_AP_CTL) Register 29-75
MXVR Asynchronous Packet Receive Buffer Start Address (MXVR_APRB_START_ADDR) Register 29-77
MXVR Asynchronous Packet Receive Buffer Current Address (MXVR_APRB_CURR_ADDR) Register 29-78
MXVR Asynchronous Packet Transmit Buffer Start Address (MXVR_APTB_START_ADDR) Register 29-79
MXVR Asynchronous Packet Transmit Buffer Current Address (MXVR_APTB_CURR_ADDR) Register 29-79
MXVR Control Message Control (MXVR_CM_CTL) Register 29-80
MXVR Control Message Receive Buffer Start Address (MXVR_CMRB_START_ADDR) Register 29-82
MXVR Control Message Receive Buffer Current Address (MXVR_CMRB_CURR_ADDR) Register 29-83
MXVR Control Message Transmit Buffer Start Address (MXVR_CMTB_START_ADDR) Register 29-84
MXVR Control Message Transmit Buffer Current Address (MXVR_CMTB_CURR_ADDR) Register 29-85
MXVR Remote Read Buffer Start Address (MXVR_RRDB_START_ADDR) Register 29-86
MXVR Remote Read Buffer Current Address (MXVR_RRDB_CURR_ADDR) Register 29-86
MXVR Pattern Registers 29-87
MXVR Pattern Data (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1) Registers 29-87

MXVR Pattern Enable (MXVR_PAT_EN_0, MXVR_PAT_EN_1) Registers 29-88
MXVR Frame Counter (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1) Registers 29-90
MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers 29-91
MXVR Block Counter (MXVR_BLOCK_CNT) Register 29-94
MXVR Clock Control (MXVR_CLK_CTL) Register 29-95
MXVR Clock/Data Recovery PLL Control (MXVR_CDRPLL_CTL) Register 29-101
MXVR Frequency Multiply PLL Control (MXVR_FMPLL_CTL) Register 29-104
MXVR Pin Control (MXVR_PIN_CTL) Register 29-106
MXVR System Clock Counter (MXVR_SCLK_CNT) Register 29-107
General Operation 29-108
Network Services Software 29-109
Network Activity Detection 29-109
Node Initialization 29-111
Initialization of Processor Pin Multiplexing 29-111
Master Mode Initialization of MXVR_CONFIG Register 29-112
Slave Mode Initialization of MXVR_CONFIG Register 29-112
Initialization of the MXVR_CLK_CTL Register 29-112
Initialization of the MXVR_ROUTING_x Registers 29-113
Initialization of the Buffer Start Address Registers 29-113

Start Up of the MXVR PLLs 29-114
Master Mode Initialization and Start Up of MXVR FMPLL and CDRPLL 29-114
Slave Mode Initialization and Start Up of MXVR CDRPLL 29-115
Enabling MXVR Output Clocks 29-116
Network Lock 29-117
Network Lock for a Master Node 29-117
Network Lock For a Slave Node 29-118
Network Initialization 29-120
Synchronous Data Routing and Muting 29-121
Synchronous Data Transmission 29-123
Synchronous Data Reception 29-125
Asynchronous Packet Transmission 29-134
Asynchronous Packet Reception 29-136
Control Message Transmission 29-138
Normal Control Message Transmission 29-142
Remote Read Control Message Transmission 29-144
Remote Write Control Message Transmission 29-146
Resource Allocate Control Message Transmission 29-148
Resource De-Allocate Control Message Transmission 29-152
Remote GetSource Control Message Transmission 29-155
Control Message Reception 29-157
Normal Control Message Reception 29-158
Remote Read and Remote Write Reception 29-159

Resource Allocate Reception 29-161
Resource De-Allocate Reception 29-162
Remote GetSource Reception 29-162
MXVR Low Power Operation 29-163
Full On Mode 29-164
Active Mode 29-166
Sleep Mode 29-167
Deep Sleep Mode 29-169
Hibernate State 29-170
Power Gating the ADSP-BF54x processor 29-171

KEYPAD INTERFACE

Interface Overview 30-1
Description of Operation 30-2
Keypad Operation 30-2
Keypad Enable/Disable 30-4
Input Keypad Matrix Programmability 30-4
Waking Up on Keypad Press 30-4
Sensitivity of Keypad Interface 30-5
Limited Multiple Key Resolution 30-5
Keypad Interrupt Modes 30-6
Implementing Press-Hold Feature 30-6
Functional Description 30-7
State Diagram 30-7
Programming Model 30-9

Keypad Registers 30-10
Keypad Control (KPAD_CTL) Register 30-10
Keypad Prescale (KPAD_PRESCALE) Register 30-13
Keypad Multiplier Select (KPAD_MSEL) Register 30-15
Keypad Row-Column (KPAD_ROWCOL) Register 30-15
Keypad Status (KPAD_STAT) Register 30-18
Keypad Software Evaluate (KPAD_SOFTEVAL) Register 30-20
Programming Examples 30-20

CAN MODULE

Overview 31-1
Interface Overview 31-2
CAN Mailbox Area 31-5
CAN Mailbox Control 31-7
CAN Protocol Basics 31-8
CAN Operation 31-10
Bit Timing 31-10
Transmit Operation 31-13
Retransmission 31-14
Single Shot Transmission 31-15
Auto-Transmission 31-16
Receive Operation 31-16
Data Acceptance Filter 31-19
Watchdog Mode 31-20
Time Stamps 31-21

Remote Frame Handling 31-22
Temporarily Disabling Mailboxes 31-23
Functional Operation 31-24
CAN Interrupts 31-24
Mailbox Interrupts 31-24
Global CAN Interrupt 31-25
Event Counter 31-28
CAN Warnings and Errors 31-29
Programmable Warning Limits 31-29
CAN Error Handling 31-30
Error Frames 31-31
Error Levels 31-33
Debug and Test Modes 31-35
Low Power Features 31-38
CAN Built-In Suspend Mode 31-39
CAN Built-In Sleep Mode 31-39
CAN Wakeup From Hibernate State 31-40
Soft Reset 31-41
CAN Registers 31-41
Global CAN Registers 31-45
Master Control (CANx_CONTROL) Registers 31-45
Global CAN Status (CANx_STATUS) Registers 31-46
CAN Debug (CANx_DEBUG) Registers 31-47
CAN Clock (CANx_CLOCK) Registers 31-47

CAN Timing (CANx_TIMING) Registers 31-48
CAN Interrupt (CANx_INTR) Registers 31-48
Global CAN Interrupt Mask (CANx_GIM) Registers 31-49
Global CAN Interrupt Status (CANx_GIS) Registers 31-49
Global CAN Interrupt Flag (CANx_GIF) Registers 31-50
Mailbox/Mask Registers 31-50
Acceptance Mask (CANx_AMxx) Registers 31-50
Mailbox Word 7 (CANx_MBxx_ID1) Registers 31-54
Mailbox Word 6 (CANx_MBxx_ID0) Registers 31-56
Mailbox Word 5 (CANx_MBxx_TIMESTAMP) Registers 31-58
Mailbox Word 4 (CANx_MBxx_LENGTH) Registers 31-59
Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers 31-61
Mailbox Control Registers 31-69
Mailbox Configuration (CANx_MCx) Registers 31-69
Mailbox Direction (CANx_MDx) Registers 31-70
Receive Message Pending (CANx_RMPx) Registers 31-71
Receive Message Lost (CANx_RMLx) Registers 31-72
Overwrite Protection/Single Shot Transmission (CANx_OPSSx) Register 31-73
Transmission Request Set (CANx_TRSx) Registers 31-74
Transmission Request Reset (CANx_TRRx) Registers 31-75
Abort Acknowledge (CANx_AAx) Registers 31-76

Transmission Acknowledge (CANx_TAx) Registers 31-77
Temporary Mailbox Disable (CANx_MBTD) Register 31-78
Remote Frame Handling (CANx_RFHx) Registers 31-78
Mailbox Interrupt Mask (CANx_MBIMx) Registers 31-79
Mailbox Transmit Interrupt Flag (CANx_MBTIFx) Registers 31-80
Mailbox Receive Interrupt Flag (CANx_MBRIFx) Registers 31-81
Universal Counter Registers 31-83
Universal Counter Configuration Mode (CANx_UCCNF) Register 31-83
Universal Counter (CANx_UCCNT) Register 31-84
Universal Counter Reload/Capture (CANx_UCRC) Register 31-84
Error Registers 31-84
Error Counter (CANx_CEC) Register 31-84
Error Status (CANx_ESR) Register 31-85
Error Counter Warning Level (CANx_EWR) Register 31-85
Programming Examples 31-85
CAN Setup Code 31-86
Initializing and Enabling CAN Mailboxes 31-87
Initiating CAN Transfers and Processing Interrupts 31-89

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers A-4
System Reset and Interrupt Control Registers A-4
Watchdog Timer Registers A-6
Real-Time Clock Registers A-6
UART0 Controller Registers A-7
UART1 Controller Registers A-8
UART2 Controller Registers A-9
UART3 Controller Registers A-10
SPI0 Controller Registers A-11
SPI1 Controller Registers A-11
SPI2 Controller Registers A-12
TWI0 Registers A-13
TWI1 Registers A-14
SPORT0 Controller Registers A-16
SPORT1 Controller Registers A-18
SPORT2 Controller Registers A-20
SPORT3 Controller Registers A-22
MXVR Registers A-24
Keypad Registers A-36
SDH Registers A-37
ATAPI Registers A-39
USB OTG Registers A-41

External Bus Interface Unit Registers A-58
DMA/Memory DMA Control Registers A-59
EPPI0 Registers A-62
EPPI1 Registers A-63
EPPI2 Registers A-64
Host DMA Registers A-65
PIXC Registers A-66
Ports Registers A-68
Timer Registers A-76
CANx Registers A-79
Handshake MDMA Control Registers A-88
NAND Flash Controller Registers A-90
Core Timer Registers A-91
Rotary Counter Registers A-91
Security Registers A-92
Processor-Specific Memory Registers A-93

TEST FEATURES

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

BYPASS – Binary Code 11111 ..... B-7
IDCODE – Binary Code 00010 ..... B-7
Boundary-Scan Register B-8

GLOSSARY

INDEX

PREFACE

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

Purpose of This Manual

ADSP-BF54x Blackfin Processor Hardware Reference provides architectural information about the ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, and ADSP-BF549 processors. This hardware reference provides architectural information about these processors and the peripherals contained within the ADSP-BF54x Blackfin packages. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see *Blackfin Processor Programming Reference.* For timing, electrical, and package specifications, see *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Intended Audience

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

What's New in This Manual

This is Revision 1.1 of the *ADSP-BF54x Blackfin Processor Hardware Reference*. This revision corrects minor typographical errors and the following issues:

- UART not half-duplex in Chapter 1, "Introduction"
- Core priority over DMA when accessing L1 SRAM in Chapter 2, "Chip Bus Hierarchy"
- Range for UNSECURED ECC SPACE in the Public OTP Memory Map in Chapter 4, "One-Time Programmable Memory"
- Internal timing requirement for DDR operation, functionality of the `MDDRENABLE` bit, sampling the `ARDY` pin when it is asserted, and the sampling edge of the `ARE` pin in Chapter 5, "External Bus Interface Unit"
- Bit settings for the internal and external triggers, `EPPIx_FS2W_LVB` register name and note, and description of `ITU_TYPE` in Chapter 15, "Enhanced Parallel Peripheral Interface"
- SESR location, software implementation of the Advanced Encryption Standard (AES), and buffer size pointed to by `pRoundKeys` in Chapter 16, "Security"
- Target address setting by elfloader utility, `MOSI` pin latching information, note on protecting the NAND boot stream, and system reset code example in Chapter 17, "System Reset and Booting"
- Arithmetic operators in PLL block diagram, note on programming the `STOPCK` bit, `CLKBUF` behavior during hibernate, and active polarity of the `EXT_WAKE` signal in Chapter 18, "Dynamic Power Management"

- Description of ATAPI_PIO_TIM_0 register in Chapter 21, “ATAPI Interface”
- Port for enabling /SPISEL3, termination of SPI TX DMA operations and comments on SPI_CTL register functionality in Chapter 22, “SPI-Compatible Port Controllers”
- TWI0_SLAVE_STAT, TWI0_SLAVE_ADDR, TWI0_MASTER_STAT0, and TWI0_MASTER_ADDR register addresses in Chapter 23, “Two-Wire Interface Controllers”
- Description of multichannel mode operation, behavior on startup when using an external clock and receiver and transmitter enable bit names standardized on RSPEN and TSPEN in Chapter 24, “SPORT Controllers”
- Notes on CAN configuration mode and CAN_GIS and CAN_GIF programming, and CANx_GIF registers marked as W1C in Chapter 31, “CAN Module”
- USB_EP_N17_RXINTERVAL and USB_EP_N17_TXCOUNT register addresses in Appendix A, “System MMR Assignments”

Technical Support

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

- Post your questions in the processors and DSP support community at EngineerZone®:
  http://ez.analog.com/community/dsp
- Submit your questions to technical support directly at:
  http://www.analog.com/support

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

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

- E-mail your questions about processors and processor applications to:
  `processor.support@analog.com` or
  `processor.china@analog.com` (Greater China support)

- In the **USA only**, call **1-800-ANALOGD** (1-800-262-5643)

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

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

Supported Processors

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

Product Information

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

Analog Devices Web Site

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

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

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

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

EngineerZone

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

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

Notation Conventions

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the IDE environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	**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.

Register Diagram Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top, followed by the short form of the name in parentheses.
- If the register is read-only (RO), write-1-to-set (W1S), or write-1-to-clear (W1C), this information appears under the name. Read/write is the default and is not noted. Additional descriptive text may follow.
- If any bits in the register do not follow the overall read/write convention, this is noted in the bit description after the bit name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name in parentheses.
- The reset value appears in binary in the individual bits and in hexadecimal to the right of the register.
- Bits marked *x* have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.
- Shaded bits are reserved.

To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.

The following figure shows an example of these conventions.

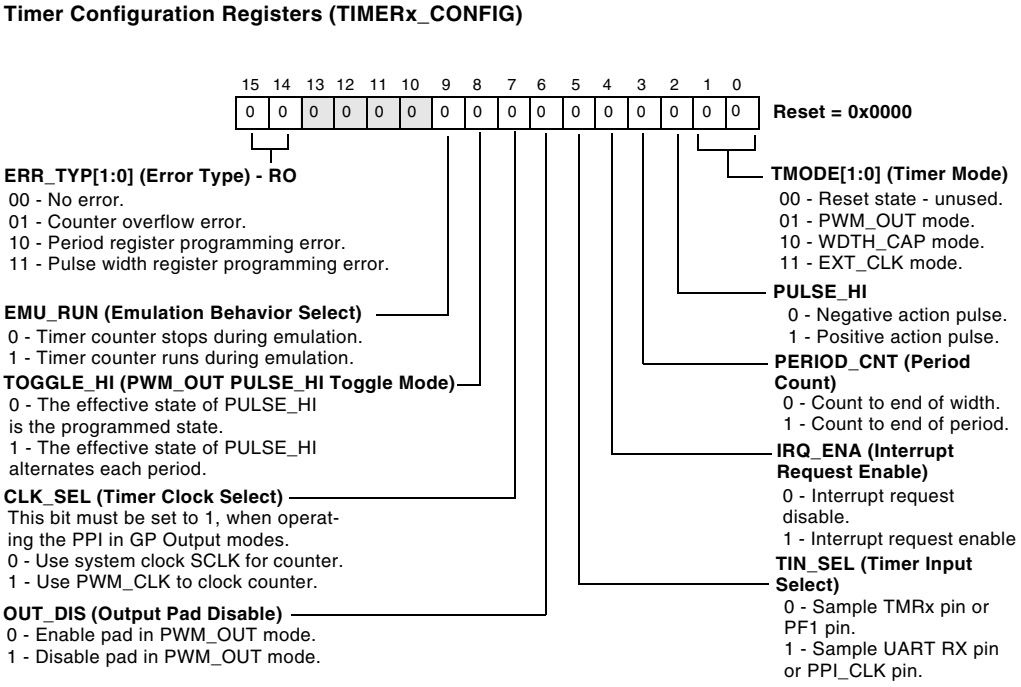


Figure 1. Register Diagram Example

1 INTRODUCTION

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

The chapter includes the following sections:

- "Peripherals" on page 1-2
- "Memory Architecture" on page 1-5
- "DMA Support" on page 1-10
- "External Bus Interface Unit" on page 1-13
- "Ports" on page 1-14
- "Two-Wire Interface" on page 1-15
- "Controller Area Network" on page 1-16
- "Enhanced Parallel Peripheral Interface (EPPI)" on page 1-17
- "SPORT Controllers" on page 1-19
- "Serial Peripheral Interface (SPI) Port" on page 1-20
- "Timers" on page 1-21

- "UART Ports" on page 1-22
- "USB On-The-Go, Dual-Role Device Controller" on page 1-23
- "ATA/ATAPI–6 Interface" on page 1-24
- "Keypad Interface" on page 1-24
- "Secure Digital (SD)/SDIO Controller" on page 1-25
- "Rotary Counter Interface" on page 1-26
- "Security" on page 1-26
- "Media Transceiver Mac Layer (MXVR)" on page 1-27
- "Real-Time Clock" on page 1-29
- "Watchdog Timer" on page 1-30
- "Clock Signals" on page 1-30

Peripherals

The processor system peripherals include combinations of:

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

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

These peripherals are connected to the core through several high band-width buses, as shown in Figure 1-1.

All of the peripherals, except for general-purpose I/O, CAN, TWI, RTC, and timers, are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces, which include external DDR1 SDRAM and asynchronous memory. Multiple on-chip buses provide enough band-width to keep the processor core running even when there is also activity on all of the on-chip and external peripherals.

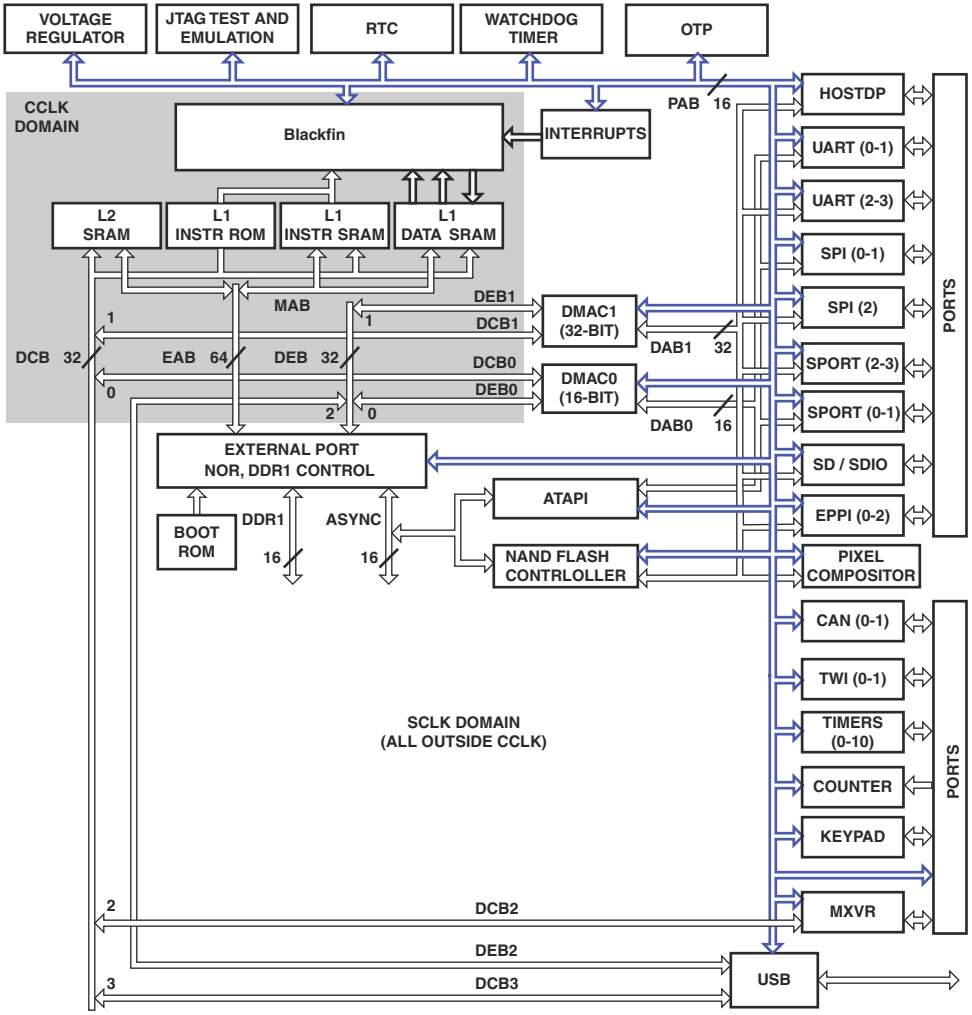


Figure 1-1. ADSP-BF54x processor Processor Block Diagram

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems. Table 1-1 shows the memory comparison for the ADSP-BF54x processor processors.

Table 1-1. Memory Configurations

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

1 This ROM is not customer configurable.

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the external bus interface unit (EBIU), provides expansion with double-data SDRAM (DDR1), flash memory, and SRAM, optionally accessing up to 516M bytes of physical memory.

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

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

Internal Memory

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

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

- L2 SRAM, providing up to 128K bytes of unified instruction and data memory, operating at one half the frequency of the core.
- 4K boot ROM that can be seen as L3 memory. It operates at full SCLK rate.

External Memory

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

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

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

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

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

NAND Flash Controller (NFC)

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

- Support for page program, page read, and block erase of NAND flash devices, with accesses aligned to page boundaries
- Error checking and correction (ECC) hardware that facilitates error detection and correction
- A single 8-bit or 16-bit external bus interface for commands, addresses and data
- Support for SLC (single-level cell) NAND flash devices unlimited in size, with page sizes of 256 and 512 bytes. Larger page sizes can be supported in software

- Capability of releasing external bus interface pins during long accesses
- Support for internal bus requests of 16- or 32-bits
- DMA engine to transfer data between internal memory and NAND flash device

I/O Memory Space

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

One-Time-Programmable (OTP) Memory

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

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

This memory is organized into 512 pages each comprised of 128 bits and equally separated into two distinct areas with privileged access dependant upon modes of operation when security features are utilized. Approximately 400 pages are available for developer use. The remaining 100 pages

are utilized for page protection bits, error correction, and ADI factory reserved areas. One area is read/write accessible at all time (public OTP memory). The second area maintains privileged access and can only be accessed (read/write) upon entry to secure mode when security features are utilized (private OTP memory).

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

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

DMA Support

ADSP-BF54x processor processors have multiple, independent DMA channels that support automated data transfers with minimal overhead for the processor core. DMA transfers can occur between the ADSP-BF54x processor processor's internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including DDR and asynchronous memory controllers.

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

All four DMA controllers use the same 32-bit DCB bus to exchange data with L1 memory. This includes L1 ROM, but excludes scratchpad memory. Fine granulation of L1 memory and special DMA buffers minimize potential memory conflicts, if the L1 memory is accessed by the core contemporaneously. Similarly, there are dedicated DMA buses between the DMAC1, DMAC0, and USB DMA controllers and the external bus interface unit (EBIU) that arbitrates DMA accesses to external memories and boot ROM.

The ADSP-BF54x processor processor DMA controllers support both one-dimensional (1D) and two-dimensional (2D) DMA transfers. DMA transfer initialization can be implemented from registers or from sets of parameters called descriptor blocks.

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

Examples of DMA types supported by the ADSP-BF54x processor processor DMA controller include:

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

In addition to the dedicated peripheral DMA channels, both the DMAC1 and the DMAC0 controllers feature two memory DMA channel pairs for transfers between the various memories of the ADSP-BF54x processor

processor system. This enables transfers of blocks of data between any of the memories—including external DDR, ROM, SRAM, and flash memory—with minimal processor intervention. Like peripheral DMAs, memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

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

Host DMA Interface

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

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

- Allows an external master to configure DMA read/write data transfers and read port status
- Uses an asynchronous memory protocol for its external interface

- Allows 8- or 16-bit external data interface to the host device
- Supports half-duplex operation
- Supports little/big endian data transfers
- Acknowledge mode allows flow control on host transactions
- Interrupt mode guarantees a burst of FIFO depth host transactions

External Bus Interface Unit

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

DDR SDRAM Controller

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

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

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

Asynchronous Controller

The asynchronous memory controller provides a configurable interface for up to four separate banks of memory or I/O devices. Each bank can be independently programmed with different timing parameters. This allows connection to a wide variety of memory devices, including SRAM, ROM, and flash EPROM, as well as I/O devices that interface with standard memory control lines. Each bank occupies a 64M byte window in the processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks are 16 bits wide, for interfacing to a range of memories and I/O devices.

Ports

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

General-Purpose I/O (GPIO)

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

After reset, all pins are in GPIO mode by default. Neither GPIO output nor input drivers are active by default. Unused pins can be left unconnected. GPIO data and direction control registers provide flexible write-1-to-set and write-1-to-clear mechanisms so that independent software threads do not need to protect against each other because of expensive read-modify-write operations when accessing the same port.

Two-Wire Interface

The ADSP-BF54x processor processor offers up to two TWI (two-wire interface) interfaces and is fully compatible with the widely used I^2C bus standard. It is designed with a high level of functionality and is compatible with multimaster, multislave bus configurations. To preserve processor bandwidth, the TWI controller can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol-related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the 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 multimaster data arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs

- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lockup
- Input filter for spike suppression
- Serial camera control bus support as specified in *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*

Controller Area Network

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

The ADSP-BF54x processor CAN controllers offer:

- 32 mailboxes (8 receive only, 8 transmit only, 16 configurable for receive or transmit)
- Dedicated acceptance masks for each mailbox
- Additional data filtering on first two bytes
- Support for both the standard (11-bit) and extended (29-bit) identifier (ID) message formats
- Support for remote frames
- Active or passive network support

- CAN wakeup from hibernation mode (lowest static power consumption mode)
- Interrupts, including: TX complete, RX complete, error, global

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

Enhanced Parallel Peripheral Interface (EPPI)

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

The following features are supported in the EPPI module.

- Programmable data length: 8, 10, 12, 14, 16, 18, and 24 bits per clock
- Bidirectional and half-duplex port
- `PPI_CLK` can be provided externally or can be generated internally
- Various framed and nonframed operating modes. Frame syncs can be generated internally or can be supplied by an external device

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

SPORT Controllers

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

- I^2S capable operation

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

Each SPI port uses three pins for transferring data: two data pins and a clock pin. An SPI chip select input pin lets other SPI devices select the processor, and seven SPI chip select output pins let the processor select other SPI devices. The SPI select pins are reconfigured, general-purpose I/O pins. Using these pins, the SPI port provides a full-duplex, synchronous serial interface, which supports both master and slave modes and multimaster environments.

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

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

Timers

There are up to two timer units in the ADSP-BF54x processor processors. Depending on the processor, one unit provides eight general-purpose programmable timers, and the other unit provides three of them. Each timer has an external pin that can be configured either as a pulse width modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths and periods of external events. These timers can be synchronized to an external clock input (to the several other associated GPIO pins) to an external clock input to the `PPI_CLK` input pin, or to the internal `SCLK`.

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

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

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

UART Ports

The ADSP-BF54x processor processor provides four full-duplex universal asynchronous receiver/transmitter (UART) ports. Each UART port provides a simplified UART interface to other peripherals or hosts, providing DMA-supported, asynchronous transfers of serial data. The UART ports include support for five to eight data bits; one or two stop bits; and none, even, or odd parity. The UART ports support two modes of operation:

- Programmed I/O

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

- Direct Memory Access (DMA)

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

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

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

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

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

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

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

The USB OTG controller provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data using a point-to-point USB connection without the need for a PC host. The USBDRC module can operate in a traditional USB peripheral-only mode as well as the host mode presented in the on-the-go (OTG) supplement to the USB 2.0 spec-

ification. In host mode, the USB module supports transfers at high-speed (480 Mbps), full speed (12 Mbps), and low speed (1.5 Mbps) rates. Peripheral-only mode supports the high and full speed transfer rates.

ATA/ATAPI–6 Interface

The ATA/ATAPI interface connects to CD/DVD and HDD drives and is ATAPI-6 compliant. The controller implements the peripheral I/O mode, the multi-DMA mode, and the Ultra DMA mode. The DMA modes enable faster data transfer and reduced host management. The ATAPI Controller supports PIO, multi-DMA, and Ultra DMA ATAPI accesses. Key features include:

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

Keypad Interface

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

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

- Supports a maximum of 8-by-8 keypad matrix
- Programmable input keypad matrix size
- Debounce filter on input signals
- Programmable debounce filter width
- Press-release/press mode supported
- Infrastructure for press-hold mode present
- Interrupt on any key pressed capability
- Multiple key pressed detection and limited multiple key resolution capability

Secure Digital (SD)/SDIO Controller

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

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

Rotary Counter Interface

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

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

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

Security

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

- Blackfin Lockbox Secure Technology incorporates a secure hardware platform for confidentiality and integrity protection of secure code and data with authenticity maintained by secure software.
- This secure platform provides:
- A secure execution mode

- Secure storage for on-chip keys
- On-chip secure ROM
- Secure RAM
- Access to code and data in the secure domain is monitored by the hardware and any unauthorized access to the secure domain is prevented.
- The secure ROM code establishes the root of trust for the secure software in the system.
- The secure RAM provides integrity protection and confidentiality for authenticated code and data.
- User-defined cipher key(s) and ID(s) and can be securely stored in the on-chip OTP memory.
- Every processor ships from the ADI factory with a unique chip ID value stored in publicly accessible OTP memory area.

Media Transceiver Mac Layer (MXVR)

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

The MXVR is fully compatible with the industry-standard standalone MOST controller devices, supporting 22.579 Mbps or 24.576 Mbps data transfer. It offers faster lock times, greater jitter immunity, a sophisticated DMA scheme for data transfers. The high-speed internal interface to the

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

core and L1 memory allows the full bandwidth of the network to be utilized. The MXVR can operate as either the network master or as a network slave.

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

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

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

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

Real-Time Clock

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

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

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

The stopwatch function counts down from a programmed value, with one second resolution. When the stopwatch is enabled and the counter underflows, an interrupt is generated.

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

Watchdog Timer

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

If configured to generate a hardware reset, the watchdog timer resets both the core and the ADSP-BF54x processor processor peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

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

Clock Signals

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

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

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip phase-locked loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (0.5x to 64x) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 8x, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

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

Dynamic Power Management

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

Full On Mode (Maximum Performance)

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

Active Mode (Moderate Dynamic Power Savings)

In the active mode, the PLL is enabled, but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. DMA access is available to appropriately configured L1 and L2 memories.

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

Sleep Mode (High Dynamic Power Savings)

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

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

Deep Sleep Mode (Maximum Dynamic Power Savings)

The deep sleep mode maximizes dynamic power savings by disabling the processor core and synchronous system clocks (CCLK and SCLK). Asynchronous systems, such as the RTC, may still be running, but cannot access internal resources or external memory. This powered-down mode can only be exited by assertion of the reset interrupt or by an asynchronous interrupt generated by the RTC. When in deep sleep mode, an RTC asynchronous interrupt causes the processor to transition to the active mode. Assertion of $\overline{\text{RESET}}$ while in deep sleep mode causes the processor to transition to the full on mode.

Hibernate State (Maximum Power Savings)

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

Voltage Regulation

The processor provides an on-chip voltage regulator that can generate internal voltage levels. The voltage regulation circuit figure in the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* shows the typical external components required to complete the power management system. The regulator controls the internal logic voltage levels and is programmable with the voltage regulator control register (`VR_CTL`) in increments of 50 mV. To reduce standby power consumption, the internal voltage regulator can be programmed to remove power to the processor core while keeping I/O power supplied. While in this state, V_{DDEXT} can still be applied, eliminating the need for external buffers. The regulator can also be disabled and bypassed at the user's discretion. For more information, see the Voltage Regulator Circuit diagram in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Boot Modes

The ADSP-BF54x processor processor has many mechanisms (listed in Table 1-2) for automatically loading internal and external memory after a reset. The boot mode is defined by four `BMODE` input pins dedicated to this purpose. There are two categories of boot modes, master and slave. In

master boot mode, the processor actively loads data from parallel or serial memories. In slave boot mode, the processor receives data from an external host device.

Table 1-2. Booting Modes

BMODE [3: 0]	Description
b#0000	Idle–no boot
b#0001	Boot from 8- or 16-bit external flash memory
b#0010	Boot from 16-bit asynchronous FIFO
b#0011	Boot from serial SPI memory (EEPROM or flash)
b#0100	Boot from SPI host device
b#0101	Boot from serial TWI memory (EEPROM/flash)
b#0110	Boot from TWI host
b#0111	Boot from UART host
b#1000	Reserved
b#1001	Reserved
b#1010	Boot from (DDR) SDRAM
b#1011	Boot from OTP memory
b#1100	Reserved
b#1101	Boot from 8- or 16-bit NAND flash memory via NFC
b#1110	Boot from 16-Bit Host DMA
b#1111	Boot from 8-Bit Host DMA

Instruction Set Description

The ADSP-BF54x processor processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to *Blackfin Processor Programming Reference* for detailed information. The instructions have been specifically tuned to provide a

flexible, densely encoded instruction set that compiles to a very small final memory size. The instruction set also provides fully featured multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. Coupled with many features more often seen on micro controllers, this instruction set is very efficient when compiling C and C++ source code. In addition, the 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 by a complete set of software and hardware development tools, including Analog Devices' emulators and the CrossCore Embedded Studio or VisualDSP++ development environment. (The emulator hardware that supports other Analog Devices processors also emulates the processor.)

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

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

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

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

2 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, how data moves through the system, and factors that determine the system organization. The chapter describes the system internal chip interfaces and discusses the system interconnects, including the interfaces between core buses and system buses.

The chapter includes the following sections:

- "Overview" on page 2-1
- "System Overview" on page 2-8
- "Peripheral Access Bus (PAB)" on page 2-15
- "DMA-Related Buses" on page 2-17
- "External Access Bus (EAB)" on page 2-24

Overview

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

Internal Interfaces

Figure 2-1 shows the processor core, on-chip peripherals, and the bus interfaces between them.

The processor core has several blocks of on-chip memory. The *L1 instruction memory* is 48K bytes SRAM plus 16K bytes that can be configured as a four-way set-associative cache or SRAM. The *L1 data memory* is 32K bytes SRAM plus 32K bytes that can be configured as a two-way set associative cache or SRAM. The *scratchpad SRAM memory* (not shown in Figure 2-1) consists of 4K bytes, which is only accessible as data SRAM (cannot be configured as cache memory). The *L1 instruction ROM memory* is factory programmed; this ROM is not customer-configurable. The *L2 SRAM memory* provides 128K bytes of unified instruction and data memory. Unlike L1 memory - which operates at the full core clock (CCLK) rate - the memory other than L1 operates at one half the frequency of the core. The *4K boot ROM* is seen as part of L3 memory. Because the boot ROM is outside the CCLK domain, this ROM operates at the system clock (SCLK) rate.

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

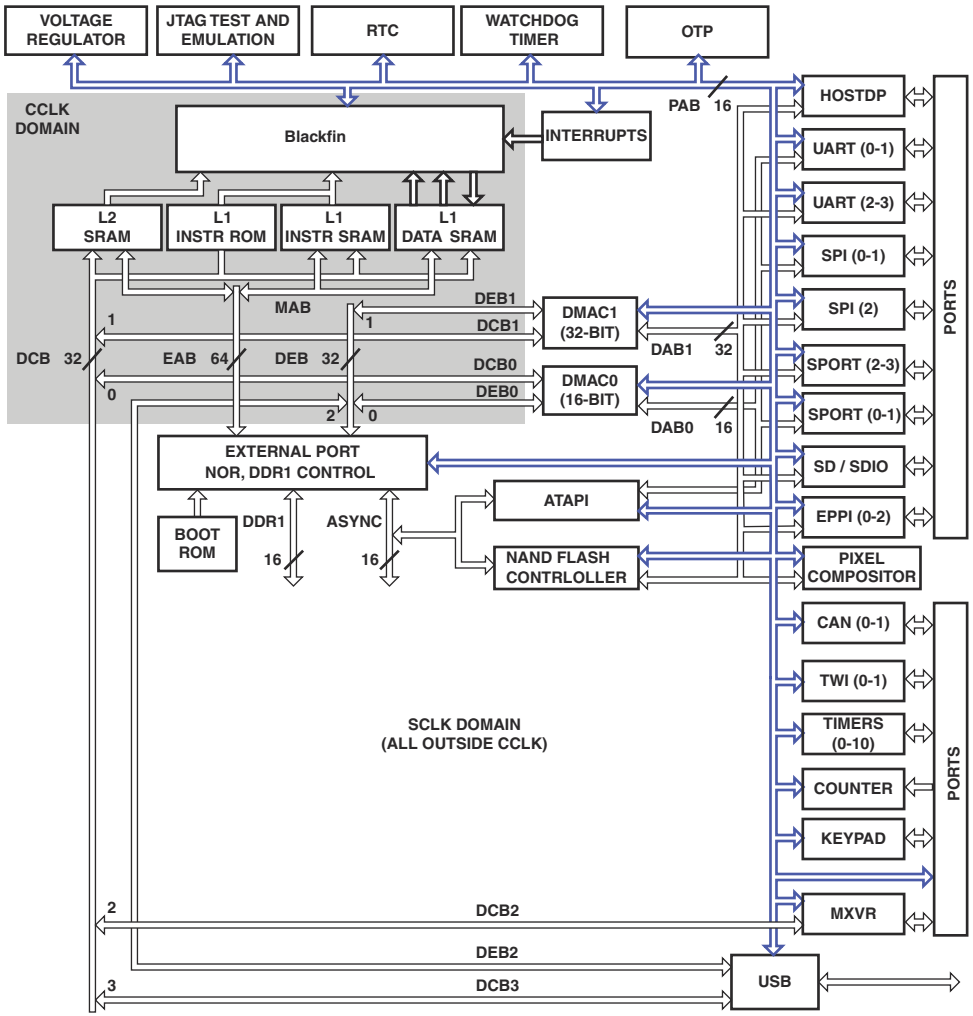


Figure 2-1. Bus Hierarchy

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

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

- 16-bit core port

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

- 64-bit core P port

  This is the processor core L1 memory access port. The P port provides the interface to the external bus interface unit (EBIU) and to the memory other than L1 through the 32-bit external access bus (EAB) and the 64-bit processor core L2 bus separately. The memory access request commands from the core are pipelined; no arbitration logic is needed in this interface. For more information, see "P Port Interface" on page 2-8.

- 32-bit core D port

  DMA controllers (DMAC0, DMAC1, USB, and MXVR) transfer data to or from core L1 memory through this port. Because there are multiple DMA controllers that can simultaneously request access to L1 memory through the 32-bit D port, interface arbitration logic is provided and described in "D Port Interface" on page 2-9.

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

- 64-bit core L2 bus

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

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

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

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

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

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

Internal Clocks

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

The peripheral access bus (PAB), the DMA access buses (DAB32 and DAB16), the external access bus (EAB), the DMA core buses (DCB0, DCB1, DCB2, and DCB3), the DMA external buses (DEB0, DEB1, and DEB2), the external port bus (EPB), and the external bus interface unit (EBIU) run at the system clock frequency (`SCLK` domain). This divider ratio is set using the `CSEL` parameter of the `PLL_DIV` register. Note that this divider must be set such that these buses run as specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*, running at a speed slower than or equal to the core clock frequency.

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

Core Bus Overview

Figure 2-2 shows a processor core block diagram that includes a processor core and L1 memory connected by internal core buses. The core bus structure between the processor core and L1 memory runs at the full core frequency (`CCLK` domain). Data loads are performed using the LD0 and

LD1 buses. The SD bus is used to perform writes. There are two address buses (DA0 and DA1) used for data fetches. The instruction address and data buses (IAB and IDB) are used to fetch instructions.

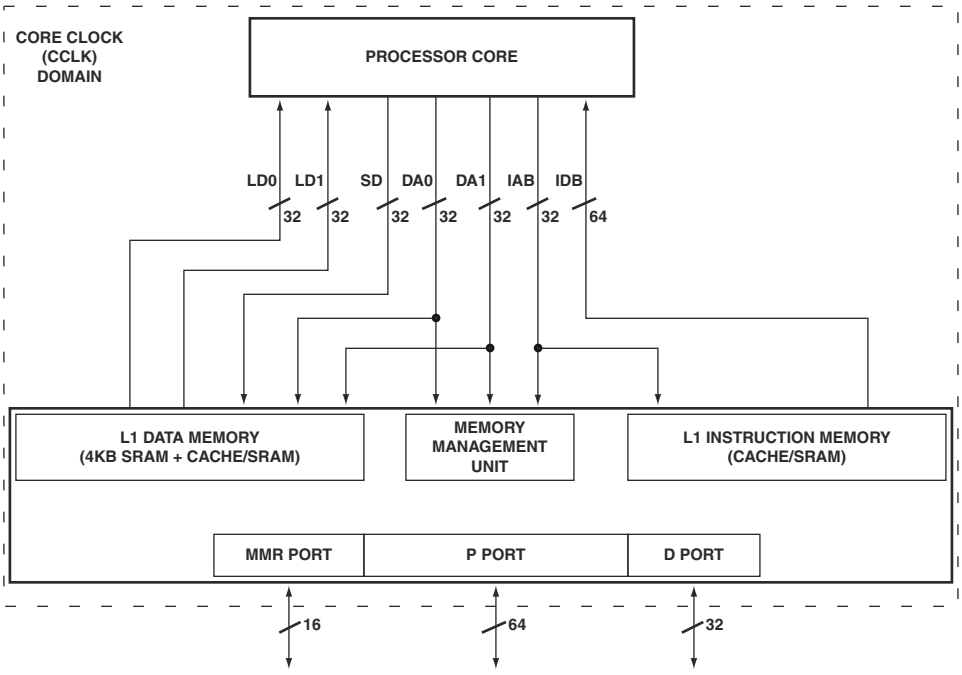


Figure 2-2. Processor Core and L1 Memory Block Diagram

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

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

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

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

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

System Overview

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

The external bus interface unit (EBIU) is the primary interface to the chip pins. Detailed information about the EBIU is discussed in "External Bus Interface Unit" on page 5-1.

P Port Interface

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

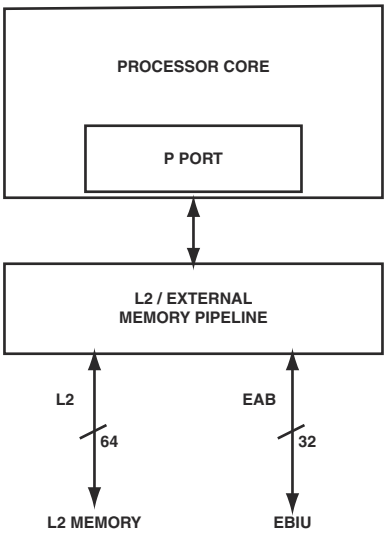


Figure 2-3. Core Interface to Memory other than L1 and the EBIU

At each CCLK cycle, the processor core can:

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

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

D Port Interface

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

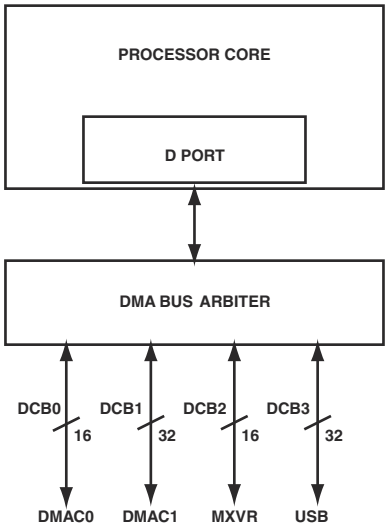


Figure 2-4. Core (A or B) Interface to DMA Controllers

The DCB buses provide the following DMA data transfers:

- The DCB0 bus supports up to 16-bit DMA data transfers between the processor core internal memory and peripheral on the DAB0 bus; or transfers between external memory and internal memory. Where internal memory is L1, a DCB bus can also support internal memory to internal memory transfers. The DCB0 bus is in the SCLK domain.
- The DCB1 bus supports up to 32-bit DMA data transfer between the processor core internal memory and peripherals on the DAB1 bus; or between external memory and internal memory. The DCB1 bus is in the SCLK domain.
- The DCB2 bus supports up to 32-bit DMA data transfer between the processor core internal memory and MXVR. The DCB2 bus is in the SCLK domain.

- The DCB3 bus supports up to 32-bit DMA data transfer between the processor core internal memory and USB. The DCB3 bus is in the SCLK domain.

Because D port access requests can come from multiple independent DMA controllers, DMA bus arbitration is necessary to resolve possible D port access conflicts. The D port interface performs DMA bus (DCB0, DCB1, DCB2, and DCB3) arbitration, converts transactions on these buses to the core DMA bus protocol, and conducts transactions over the core DMA buses to L1 memory or over a separate bus to the memory other than L1. For more information on DMA priority arbitration, see "DCB Arbitration" on page 2-20.

On-Chip L2 Interface

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

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

An L2 access requires two CCLK cycles for the access itself, plus any latency involved in the operation (see Table 2-2 on page 2-14). L2 interface control logic is clocked at the core frequency (CCLK clock domain). The system DMA access request comes from the DCB0, DCB1, DCB2, and the

DCB3 busses, which run at system clock frequency (SCLK domain). The interface circuit synchronizes the DCB buses to the core clock domain and converts them to system L2 bus protocol.

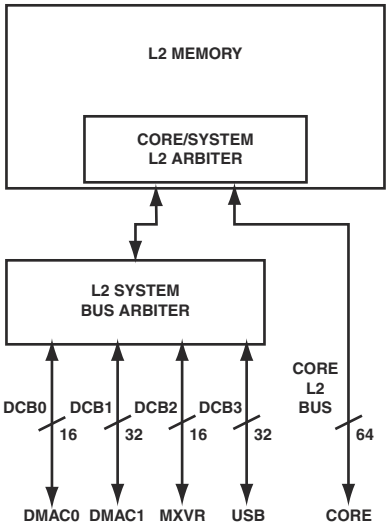


Figure 2-5. L2 Bus Interfaces

As shown in Figure 2-5 on page 2-12, there are several arbitration stages in the interface:

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

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

- Arbitration for system L2 port access request is based on a programmable priority scheme. After reset, the following fixed priority is maintained: DCB2 (MXVR) > DCB0 (DMAC0) >DCB1 (DMAC1) > DCB3 (USB) for L2 accesses through the system L2

bus. The priorities between the DCB0 bus and DCB1 bus is programmable. This can be using the L2DMAPRO bit in the SYSCR register.

- When both the processor core L2 bus and the system L2 bus attempt to access the same bank at the same time, bank arbitration is required. Table 2-1 shows the L2 access bus arbitration.

Table 2-1. L2 Interface Bus Arbitration

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

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

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

For more details about DMA data transfer latency and DMA traffic control/optimization, see “DMA Performance” on page 7-50.

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

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

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

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

Peripheral Access Bus (PAB)

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

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

PAB Performance

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

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

The PAB has a maximum frequency of SCLK.

PAB Agents (Masters, Slaves)

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. System MMR addresses are listed in "System MMR Assignments" on page A-1.

The slaves on the PAB bus are:

- Event Controller
- Clock and Power Management Controller
- Watchdog Timer
- Real Time Clock
- Timer 0–10
- SPORT 0–3
- SPI 0–2
- General-Purpose Input/Output (GPIOs)
- UART 0–3
- ATAPI
- EPPI0-2
- Pixel Compositor
- Secure Digital Host (SDH)
- USB
- MXVR
- TWI 0–1
- CAN 0–1
- Asynchronous Memory Controller (AMC)
- DDR SDRAM Controller (DDC)
- DMA Controller 0–1 (DMAC0 and DMAC1)

DMA-Related Buses

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

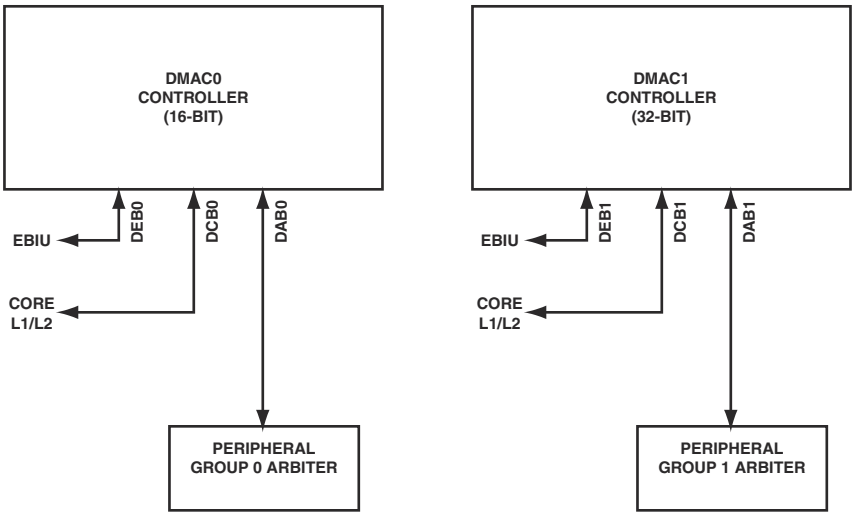


Figure 2-6. DMA Bus Connection and Arbitration Block Diagram

The 32/16-bit DAB bus provides DMA between the peripherals and L1/L2 internal memory through the DCB bus, or between peripherals and external memories through the DEB bus. A central DMA controller keeps track of DMA addresses and mediates the transfers. DMA is handled identically for 8-, 16- and 32-bit data sizes. The maximum bandwidth for any individual 16-bit peripheral is one 16-bit word transferred for every two SCLK cycles. Peripherals that are capable of 32-bit DMA (and also configured for 32-bit mode) can transfer up to one 32-bit word every two SCLK cycles.

Peripheral DMA

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

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

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access through one of the centralized DMA controllers are masters on these buses, as shown in Table 2-3 on page 2-19 and Table 2-4 on page 2-20. A single arbiter supports a programmable priority arbitration policy for access to each DAB.

When two or more DMA master channels are actively requesting a DAB, bus utilization is considerably higher due to the DAB's pipelined design. Bus arbitration cycles are concurrent with the previous DMA access data cycles. The MXVR and USB peripherals have their own DMA channels and are not part of the DAB.

DAB Arbitration

There are two centralized DMA controllers in the system which together support 14 peripherals and four memory DMA channels. 32 DMA channels and bus masters support these devices, with eight channels being assigned to memory DMA, and the remaining 24 channels being assigned to peripheral DMA. The memory DMA channels can transfer data

between L1, L2, and external memory. The peripheral DMA controllers can transfer data between peripherals and internal (L1/L2) or external memory.

The DAB buses are implemented as two separate bus systems each interfacing to a DMA controller and a fixed set of peripheral DMA bus masters. DAB0 offers 16 bits of data transfer per SCLK cycle and DAB1 offers 32 bits of data transfer per SCLK cycle. Arbitration of channels on the DAB bus is programmable within each centralized DMA controller. Table 2-3 and Table 2-4 show the default arbitration priority of each DMA controller.

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

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

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

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

DCB Arbitration

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

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

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

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

The priority order for DMAC0 and DMAC1 may be swapped. Table 2-5 describes the priority configuration for L1 accesses, which is defined by the `CDMAPRIO` bit of the `SYSCR` register. For L2 accesses, the `L2DMAPRIO` bit in `SYSCR` is used in the same way. For more information, see "System Reset Configuration (SYSCR) Register" on page 17-105.

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

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

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

DEB Arbitration

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

The MXVR does not have DMA access to external memory.

In the event of simultaneous requests to external memory, access is granted based on a programmable arbitration scheme. This priority can be changed by using the `DEB_ARB_PRIORITY` bits in the `EBIU_DDRQUE` register. For off-chip memory, the core has priority over the DEB buses by default. However, the priorities of the specific DMA bus with respect to the core can be changed for both synchronous and asynchronous accesses. The complete arbitration at the EBIU is described in "External Bus Interface Unit" on page 5-1.

DAB, DCB, and DEB Performance

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

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers. DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel.

Memory DMA transfers can result in repeated accesses to the same memory location. Because the memory DMA controller has the potential of simultaneously accessing on-chip and off-chip memory, considerable throughput can be achieved. The throughput rate for an on-chip/off-chip memory access is limited by the slower of the two accesses.

In the case where the transfer is from on-chip to on-chip memory or from off-chip to off-chip memory, the burst accesses cannot occur simultaneously. The transfer rate is then determined by adding each transfer plus an additional cycle between each transfer.

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

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

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

Source	Destination	Approximate SCLKs for n Words (Max word size 32-bits) (From Start of DMA to Interrupt at End)
16-bit DDR SDRAM	L1 Data memory	n + 14
L1 Data memory	16-bit DDR SDRAM	n + 11
16-bit Async memory	L1 Data memory	xn + 12, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 Data memory	16-bit Async memory	xn + 9, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
16-bit DDR SDRAM	16-bit DDR SDRAM	10 + (17n/7)
16-bit Async memory	16-bit Async memory	10 + 2xn, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 Data memory	L1 Data memory	2n + 12

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

External Access Bus (EAB)

The external access bus (EAB) provides a way for the processor core and the Memory DMA controller to directly access off-chip memory and high throughput memory-to-memory DMA transfers. The EAB supports single-word accesses of either 8-bit, 16-bit, or 32-bit data types. The EAB operates at the system clock rate.

EAB/DEB Arbitration

Arbitration for use of external memory interface resources (DDR or ASYNC) is required because of possible contention between the potential masters of these resources. A fixed-priority arbitration scheme is used to arbitrate between EAB accesses and DEB accesses, with core accesses winning by default. For more details on arbitration, see "External Bus Interface Unit" on page 5-1. For information on external memory interface resources, see "DDR SDRAM Memory Interface" on page 5-18 or "Asynchronous Memory Interface" on page 5-53.

EAB/DEB Performance

The EAB supports single-word accesses of 8-bit, 16-bit, 32-bit, or 64-bit data types. The EAB operates at the same frequency as the PAB and the DAB, up to the maximum SCLK frequency specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

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

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

Source	Destination	Approximate SCLKs For n 16-bit Words (From Start of DMA to Interrupt at Rnd)	Approximate SCLKs For n 32-bit Words (From Start of DMA to Interrupt at end)[1]
16-bit DDR SDRAM	L1 Data memory	n + 14	2n + 14
L1 Data memory	16-bit DDR SDRAM	n + 14	2n + 14

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

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

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

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

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

3 MEMORY

This chapter includes the following sections:

- "Memory Architecture" on page 3-2
- "Instruction Test Registers" on page 3-23
- "L1 Data Memory" on page 3-27
- "Data Test Registers" on page 3-42
- "On-Chip Level 2 (L2) Memory" on page 3-47
- "One Time Programmable Memory" on page 3-49
- "External Memory" on page 3-50
- "Memory Protection and Properties" on page 3-51
- "Memory Transaction Model" on page 3-69
- "Load/Store Operation" on page 3-70
- "Working With Memory" on page 3-76
- "Terminology" on page 3-79

Memory Architecture

The ADSP-BF54x processor processor supports a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories are located on the chip and provide faster access. Level 2 (L2) memories are on-chip memory systems (which are farther from the core) and typically have longer access latencies. The faster L1 memories, which include instruction SRAM and instruction ROM, data, and scratchpad memory as part of the Blackfin core are accessed in a single cycle. The L2 memories, which include an on-chip SRAM and off-chip synchronous and asynchronous devices, provide much higher memory space with higher latency.

The ADSP-BF54x processor processor has a unified 4G byte address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, 272M byte of address space is dedicated to internal, on-chip resources. The ADSP-BF54x processor processor populates portions of this internal memory space with:

- L1 and L2 static random access memories (SRAM)
- L1 instruction ROM (IROM)
- A set of memory-mapped registers (MMRs)
- A boot read-only memory (ROM)

A portion of the internal L1 SRAM can also be configured to run as cache. The ADSP-BF54x processor processor also provides support for an external memory space that includes asynchronous memory space and DDR space. See Chapter 5, External Bus Interface Unit for a detailed discussion of each of these memory regions and the controllers that support them.

The diagram in Figure 3-1 on page 3-4 provides an overview of the ADSP-BF54x processor system memory map. Note that the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

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

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

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

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

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

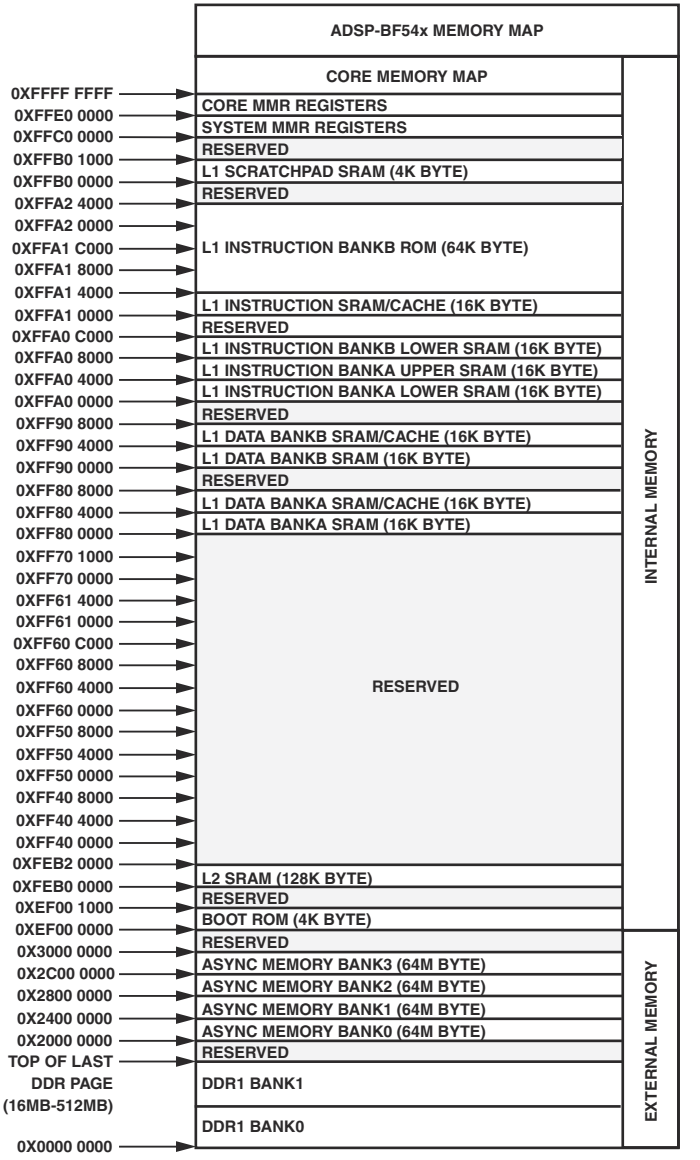


Figure 3-1. Memory Map

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

Internal Memory

The ADSP-BF54x processor L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on chip. The ADSP-BF54x processor processor supports this memory architecture for applications that require direct control over access time.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both high performance and a simple programming model. Caches eliminate the need to explicitly manage data movement into and out of the L1 memories. Code can be ported to or developed for the ADSP-BF54x processor processor quickly without requiring performance optimization for the memory organization.

Each core's L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM access at processor clock rate (CCLK) for critical DSP algorithms and fast context switching

- Instruction and data cache options for microcontroller code, excellent high-level language (HLL) support, and ease of programming cache control instructions, such as `PREFETCH` and `FLUSH`
- Memory protection

The L1 memories operate at the core clock frequency (CCLK).

Overview of L1 Instruction SRAM

The 64K byte L1 instruction SRAM consists of a dedicated 48K byte SRAM plus an additional 16K byte bank which can be configured as either SRAM or cache. The upper 16K byte, L1 instruction memory can be configured as a 4-way set-associative cache (see Figure 3-4 on page 3-15). Consequently, instructions can be brought into four different ways of cache, decreasing the frequency of cache-line replacements and increasing overall performance. When the upper 16K byte of L1 memory is configured as a cache, individual ways or lines of L1 instruction cache can be locked down, allowing further control over the location of time-critical code. The cache-locking concept is explained further in "Instruction Cache Locking by Way" on page 3-21. When configured as SRAM, each of the four 16K byte banks of memory is broken into 4K byte sub-banks which can be independently accessed by the processor and DMA. For more information about L1 instruction SRAM, see "L1 Instruction SRAM" on page 3-11.

Overview of L1 Instruction ROM

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

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

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

Overview of L1 Data SRAM

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

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

If configured as SRAM, each of the two upper 16K byte banks of memory is broken into four 4K byte sub-banks which can be independently accessed by the processor and DMA. For more information about L1 data SRAM, see “L1 Data SRAM” on page 3-30.

Overview of Scratchpad Data SRAM

The processor provides a dedicated 4K byte bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache or targeted by DMA.

Typical applications use the scratchpad data memory where speed is critical. For example, the user and supervisor stacks should be mapped to the scratchpad memory for the fastest context switching during interrupt handling.

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

Overview of On-Chip L2

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

L1 Instruction Memory

L1 instruction memory consists of a combination of dedicated SRAM and banks which can be configured as SRAM or cache. For the 16K byte bank that can be either cache or SRAM, control bits in the `IMEM_CONTROL` register can be used to organize all four sub-banks of the L1 instruction memory as any of the following:

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

L1 instruction memory can be used only to store instructions.

Instruction Memory Control Register (IMEM_CONTROL)

The instruction memory control (IMEM_CONTROL) register contains control bits for the L1 instruction memory. By default after reset, cache and cacheability protection lookaside buffer (CPLB) address checking is disabled (see "L1 Instruction Cache" on page 3-13).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRUPRIO bits (see "ICPLB Data Registers (ICPLB_DATAx)" on page 3-59) are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPLB_LRUPRIO bits, and then on LRU (least recently used). See "Instruction Cache Locking by Line" on page 3-20 for complete details. This bit must be 0 to allow the state of the CPLB_LRUPRIO bits to be stored when new lines are cached.

L1 Instruction Memory Control Register (IMEM_CONTROL)

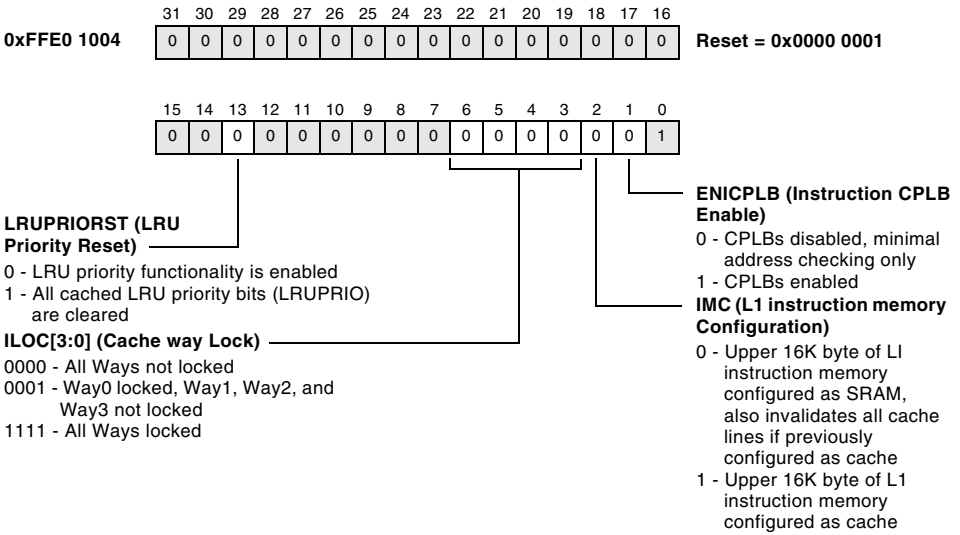


Figure 3-2. L1 Instruction Memory Control Register

The ILOC[3:0] bits provide a useful feature only after code is manually loaded into cache. See "Instruction Cache Locking by Way" on page 3-21. These bits specify which ways to remove from the cache replacement policy. This has the effect of locking code present in non-participating ways. Code in non-participating ways can still be removed from the cache using an IFLUSH instruction. If an ILOC[3:0] bit is 0, the corresponding way is not locked and that way participates in cache replacement policy. If an ILOC[3:0] bit is 1, the corresponding way is locked and does not participate in cache replacement policy.

The IMC bit reserves a portion of L1 instruction SRAM to serve as cache. Note: Reserving memory to serve as cache does not alone enable memory other than L1 accesses to be cached. CPLBs must also be enabled using the EN_ICPLB bit and the CPLB descriptors (ICPLB_DATAx and ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.

Reserving memory to serve as cache does not alone enable memory other than L1 accesses to be cached. CPLBs must also be enabled using the EN_ICPLB bit and the CPLB descriptors (ICPLB_DATAx and ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.

Instruction CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception to the processor whenever it attempts to fetch an instruction from:

- Reserved (non populated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using this bit prior to updating their descriptors (DCPLB_DATAx and DCPLB_ADDRx registers). Note since load store ordering is weak (see "Ordering of Loads and Stores" on page 3-72), disabling of CPLBs should be proceeded by a CSYNC.

When enabling or disabling cache or CPLBs, immediately follow the write to IMEM_CONTROL with a SSYNC to ensure proper behavior.

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

L1 Instruction SRAM

The ADSP-BF54x processor processor core reads the instruction memory through the 64-bit-wide instruction-fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

The DAGs cannot access L1 instruction memory directly. A DAG reference to instruction memory SRAM space generates an exception. (For information about DAG addressing, see *Blackfin Processor Programming Reference*.)

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

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

Table 3-1. L1 Instruction Memory Sub-banks

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

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

Figure 3-3 on page 3-14 describes the bank architecture of the L1 instruction memory. As the figure shows, each 16K byte bank is made up of four 4K byte sub-banks.

L1 Instruction Cache

The L1 instruction memory may also be configured as a flexible, 4-way set-associative instruction 16K byte cache. To improve the average access latency for critical code sections, each way of the cache can be locked independently. When the memory is configured as cache, it cannot be accessed directly.

When cache is enabled, only memory pages specified as cacheable by cacheability protection lookaside buffers (CPLBs) are cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in "Memory Protection and Properties" on page 3-51.

Figure 3-4 on page 3-15 shows the overall Blackfin processor instruction cache organization.

Cache Lines

As shown in Figure 3-4, the cache consists of a collection of cache lines. Each cache line is made up of a tag component and a data component:

- The tag component incorporates a 20-bit address tag, least recently used (LRU) bits, a valid bit, and a line lock bit.
- The data component is made up of four 64-bit words of instruction data.
- The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

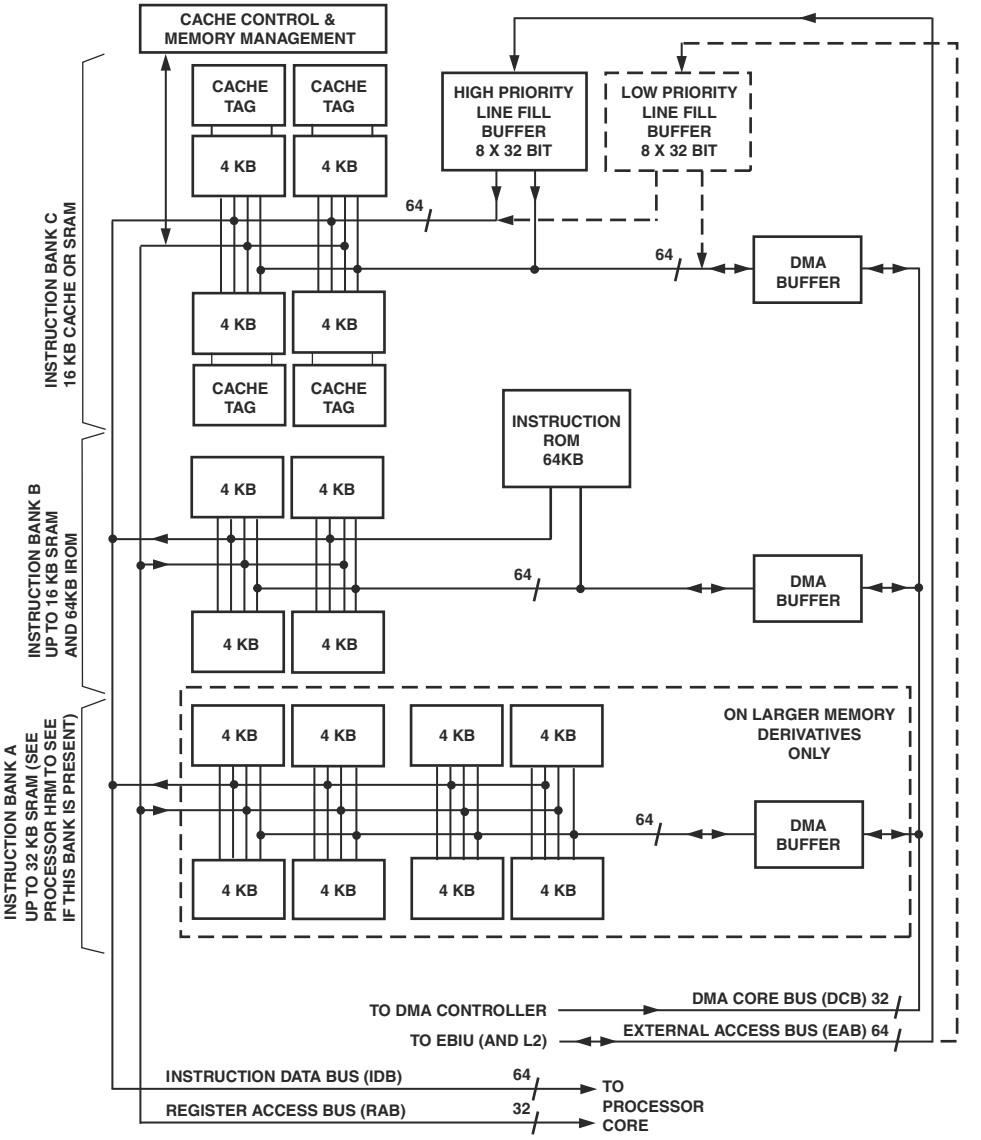


Figure 3-3. L1 Instruction Memory Bank Architecture

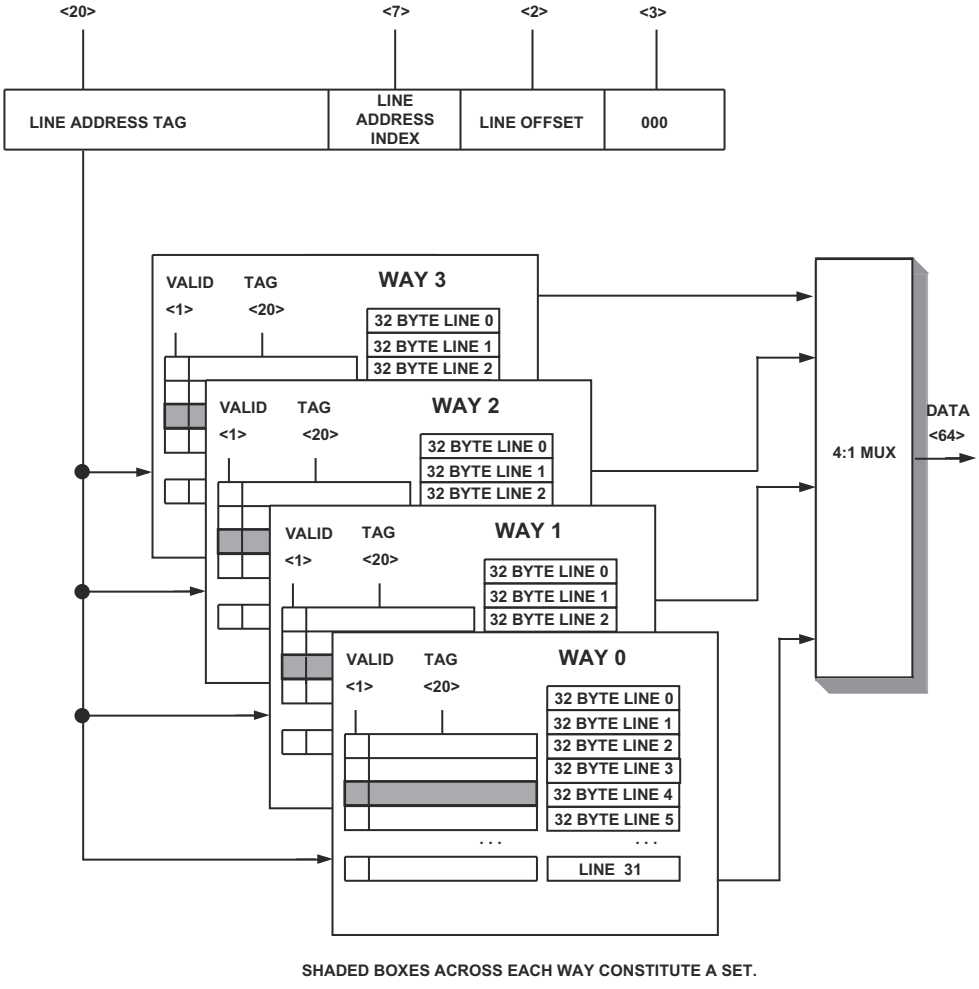


Figure 3-4. Instruction Cache Organization by Subbank

The address tag consists of the upper 18 bits plus bits 11 and 10 of the physical address. Bits 12 and 13 of the physical address are not part of the address tag. Instead, these bits are used to identify the 4K byte memory sub-bank targeted for the access.

The LRU bits are part of an LRU algorithm used to determine which cache line should be replaced if a cache miss occurs.

The valid bit indicates the state of a cache line. A cache line is always valid or invalid:

- Invalid cache lines have their valid bit cleared, indicating the line is ignored during an address-tag compare operation.
- Valid cache lines have their valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.

The tag and data components of a cache line are illustrated in Figure 3-5.

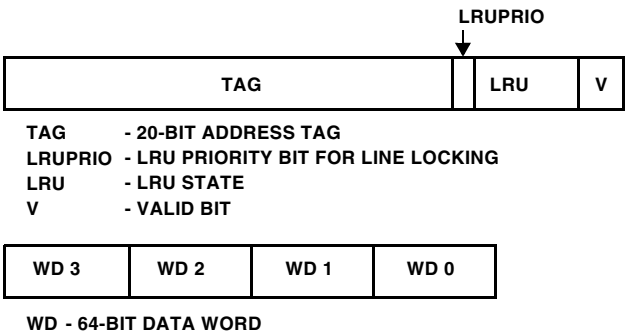


Figure 3-5. Cache Line – Tag and Data Portions

Cache Hits and Misses

A cache hit occurs when the address for an instruction-fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction-fetch address to the address tags of valid lines currently stored in a cache set. The cache set is selected, using bits 9 through 5 of the instruction-fetch address. If the address-tag compare operation results in a match, a cache hit occurs. If the address-tag compare operation does not result in a match, a cache miss occurs.

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

When a cache miss occurs, the instruction memory unit generates a cache line-fill access to retrieve the missing cache line from memory that is external to the core. The address for the on-chip L2 or external memory access is the address of the target instruction word. When a cache miss occurs, the core halts until the target instruction word is returned from on-chip L2 or external memory.

Cache-Line Fills

A cache-line fill consists of fetching 32 bytes of data from memory. The operation starts when the instruction memory unit requests a line-read data transfer (a burst of four 64-bit words of data) on its on-chip L2 or external read-data port. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the on-chip L2 or external memory returns the target instruction word first. After it has returned the target instruction word, the next three words are fetched in sequential address order. This fetch wraps around if necessary, as shown in Table 3-2.

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

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

Line-Fill Buffer

As the new cache line is retrieved from on-chip L2 or external memory, each 64-bit word is buffered into one of two four-entry line-fill buffer before it is written to a 4K byte memory bank within L1 memory. The line-fill buffer allows the core to access the data from the new cache line as the line is being retrieved from on-chip L2 or external memory, rather than having to wait until the line is written into the cache.

Two separate line-fill buffers are provided to allow a load from slow external memory to continue without causing jumps to higher speed on-chip memory other than L1 to stall. The CPLB_MEMLEV bit in the memory pages CPLBs determines which line buffer is used. See "Memory Protection and Properties" on page 3-51.

Cache-Line Replacement

When the instruction memory unit is configured as cache, bits 9 through 5 of the instruction fetch address are used as the index to select the cache set for the tag-address compare operation. If the tag-address compare operation results in a cache miss, the valid bits for the selected set are examined by a cache-line replacement unit to determine the entry to use for the new cache line, that is, whether to use Way0, Way1, Way2, or Way3 (see Figure 3-4 on page 3-15).

The cache-line replacement unit first checks for invalid entries (that is, entries having its valid bit cleared). If only a single invalid entry is found, that entry is selected for the new cache line. If multiple invalid entries are found, the replacement entry for the new cache line is selected based on the following priority:

- Way0 first
- Way1 next
- Way2 next
- Way3 last

For example:

- If Way3 is invalid and Ways0, 1, 2 are valid, Way3 is selected for the new cache line.
- If Ways0 and 1 are invalid and Ways2 and 3 are valid, Way0 is selected for the new cache line.
- If Ways2 and 3 are invalid and Ways0 and 1 are valid, Way2 is selected for the new cache line.

When no invalid entries are found, the cache replacement logic uses an LRU algorithm.

Instruction Cache Management

The system DMA controller and the core DAGs cannot access the instruction cache directly. By a combination of instructions and the use of core MMRs, it is possible to initialize the instruction tag and data arrays indirectly and provide a mechanism for instruction cache test, initialization, and debug.

The coherency of instruction cache must be explicitly managed. To accomplish this and ensure that the instruction cache fetches the latest version of any modified instruction space, invalidate instruction cache line entries, as required.

For more information, see "Instruction Cache Invalidation" on page 3-22.

Instruction Cache Locking by Line

The CPLB_LRUPRIO bits in the ICPLB_DATAx registers (see "Memory Protection and Properties" on page 3-51) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line's tag. It is then used in conjunction with the LRU (least recently used) policy to determine which way is victimized when all cache ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either "low" or "high" importance. In a modified LRU policy, a high can replace a low, but a low cannot replace a high. If all ways are occupied by highs, an otherwise cacheable low will still be fetched for the core, but will not be cached. Fetched highs seek to replace unoccupied ways first, then least recently used lows next, and finally other highs using the LRU policy. Lows can only replace unoccupied ways or other lows, and do so using the LRU policy. If *all* previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the LRUPRIRST bit in the IMEM_CONTROL register (see "Instruction Memory Control Register (IMEM_CONTROL)" on page 3-9).

Instruction Cache Locking by Way

The instruction cache has four independent lock bits (`ILOC[3:0]`) that control each of the four ways of the instruction cache. When the cache is enabled, L1 instruction memory has four ways available. Setting the lock bit for a specific way prevents that way from participating in the LRU replacement policy. Thus, a cached instruction, with its way locked, can only be removed using an `IFLUSH` instruction, or "backdoor" MMR assisted manipulation of the tag array.

An example sequence is provided to demonstrate how to lock down Way0:

- If the code of interest may already reside in the instruction cache, invalidate the entire cache first (for an example, see "Instruction Cache Invalidation" on page 3-22).
- Disable interrupts, if required, to prevent Interrupt Service Routines (ISRs) from potentially corrupting the locked cache.
- Set the locks for the other ways of the cache by setting `ILOC[3:1]`. Only Way0 of the instruction cache can now be replaced by new code.
- Execute the code of interest. Any cacheable exceptions, such as exit code, traversed by this code execution are also locked into the instruction cache.
- Upon exit of the critical code, clear `ILOC[3:1]`, and set `ILOC[0]`. The critical code (and the instructions which set `ILOC[0]`), are now locked into Way0.
- Re-enable interrupts, if required.

If all four ways of the cache are locked, then further allocation into the cache is prevented.

Instruction Cache Invalidation

The instruction cache can be invalidated by an address, cache line, or a complete cache. The IFLUSH instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the P registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated.

In the following example, the P2 register contains the address of a valid memory location. If this address is brought into cache, the corresponding cache line is invalidated after the execution of this instruction.

Example of ICACHE instruction:

```
iflush [ p2 ] ;   /* Invalidate cache line containing address
that P2 points to */
```

Because the IFLUSH instruction is used to invalidate a specific address in the ADSP-BF54x processor processor memory map, it is impractical to use this instruction to invalidate an entire bank of cache. A second, faster technique can be used to invalidate an entire cache bank directly. This second technique directly invalidates valid bits by setting the invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (ITEST_COMMAND and ITEST_DATA[1:0]) are available to allow arbitrary read/write of all cache entries directly.

For invalidating the complete instruction cache, a third method is available. By clearing the IMC bit in the IMEM_CONTROL register (see Figure 3-2 on page 3-9), all valid bits in the instruction cache are set to the invalid state. A second write to the IMEM_CONTROL register to set the IMC bit then configures the instruction memory as cache again. An SSYNC should be run before invalidating the cache and a CSYNC should be inserted after each of these operations.

Instruction Test Registers

The Instruction test registers allow arbitrary read/write of all L1 cache entries directly. They make it possible to initialize the instruction tag and data arrays and to provide a mechanism for instruction cache test, initialization, and debug.

When the instruction test command register (ITEST_COMMAND) is used, the L1 cache data or tag arrays are accessed, and data is transferred through the instruction test data registers (ITEST_DATA[1:0]). The ITEST_DATAx registers contain either the 64-bit data that the access is to write to or the 64-bit data that was read during the access. The lower 32 bits are stored in the ITEST_DATA[0] register, and the upper 32 bits are stored in the ITEST_DATA[1] register. When the tag arrays are accessed, ITEST_DATA[0] is used. Graphical representations of the ITEST registers begin with Figure 3-6 on page 3-24.

The ITEST registers are described in Table 3-3.

Access to these registers is possible only in supervisor or emulation mode. When writing to ITEST registers, always write to the ITEST_DATAx registers first, then the ITEST_COMMAND register. When reading from ITEST registers, reverse the sequence—read the ITEST_COMMAND register first, then the ITEST_DATAx registers.

Table 3-3. ITEST Registers

Name	Description/ Refer to
ITEST_COMMAND	Instruction test command register For more information, see “ITEST_COMMAND Register” on page 3-24.
ITEST_DATA1	Instruction test data 1 register For more information, see “ITEST_DATA1 Register” on page 3-25.
ITEST_DATA0	Instruction test data 0 register For more information, see “ITEST_DATA0 Register” on page 3-26.

ITEST_COMMAND Register

When the instruction test command register (ITEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the instruction test data registers (ITEST_DATA[1:0]).

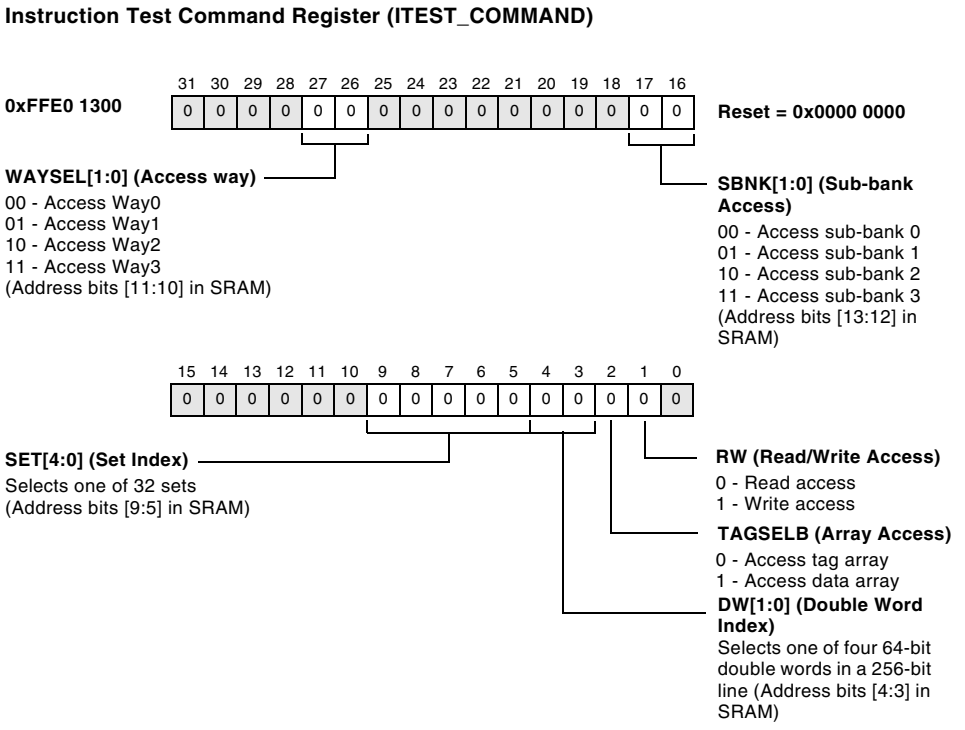


Figure 3-6. Instruction Test Command Register

ITEST_DATA1 Register

Instruction test data registers (ITEST_DATA[1:0]) are used to access L1 cache data arrays. They contain either the 64-bit data that the access is to write to or the 64-bit data that the access is to read from. The instruction test data 1 register (ITEST_DATA1) stores the upper 32 bits.

Instruction Test Data 1 Register (ITEST_DATA1)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 3-13.

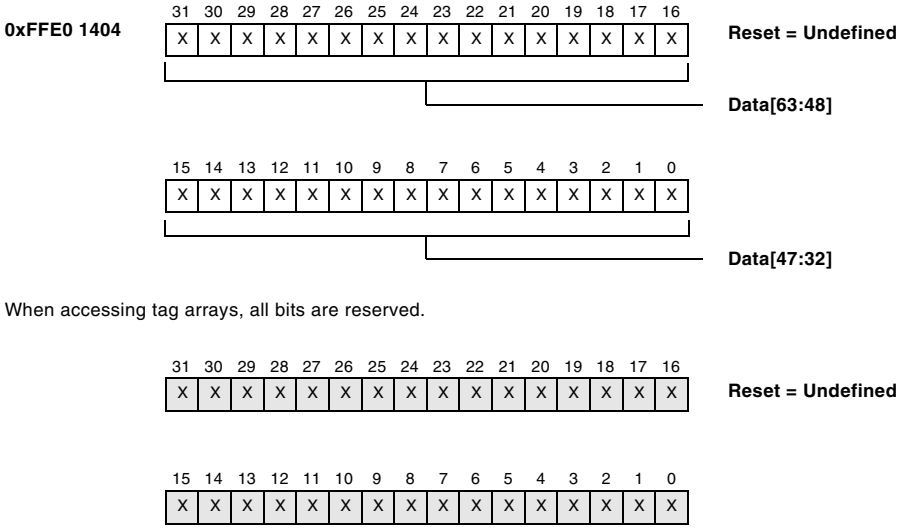


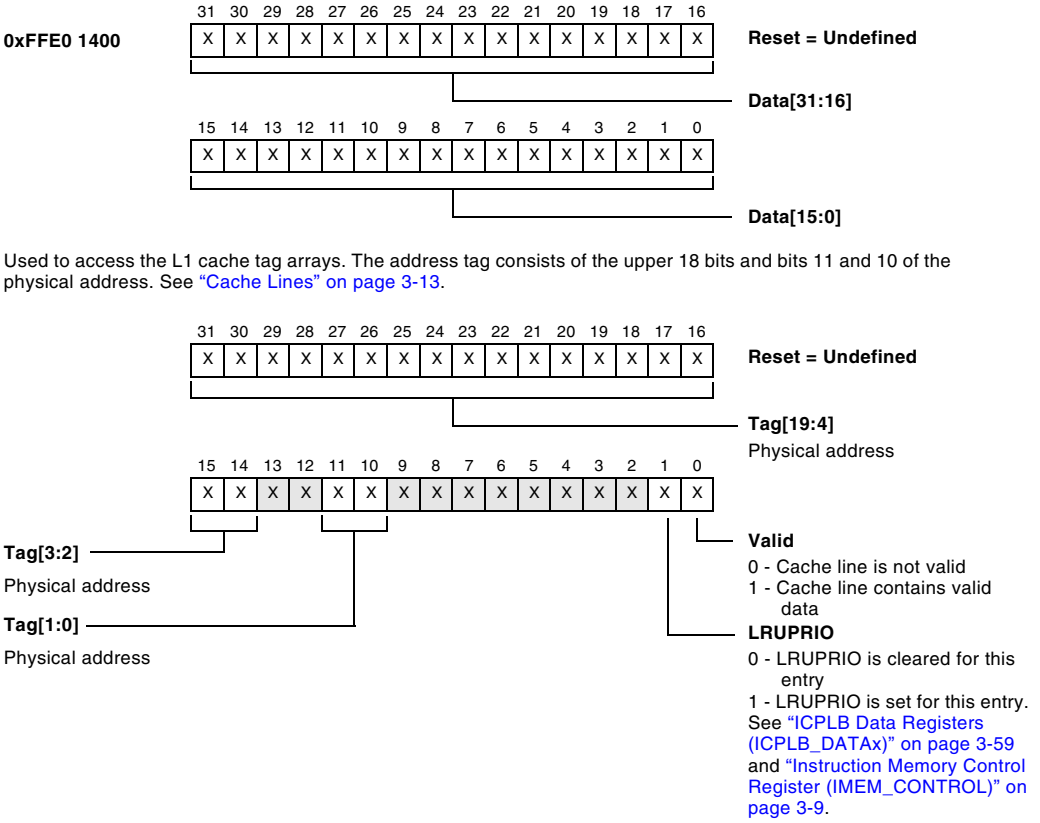
Figure 3-7. Instruction Test Data 1 Register

ITEST_DATA0 Register

The instruction test data 0 register (ITEST_DATA0) stores the lower 32 bits of the 64-bit data to be written to or read from by the access. The ITEST_DATA0 register is also used to access tag arrays. This register also contains the valid and dirty bits, which indicate the state of the cache line.

Instruction Test Data 0 Register (ITEST_DATA0)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the lower 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 3-13.



Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bits 11 and 10 of the physical address. See "Cache Lines" on page 3-13.

Figure 3-8. Instruction Test Data 0 Register

L1 Data Memory

The L1 data SRAM/cache is constructed from single-ported subsections, but organized to reduce the likelihood of access collisions. This organization results in apparent multiported behavior. When there are no collisions, this L1 data traffic could occur in a single core clock cycle:

- Two 32-bit DAG loads
- One pipelined 32-bit DAG store
- One 64-bit DMA I/O
- One 64-bit cache fill/victim access

Although L1 data memory can be used to store instructions, instructions cannot execute directly from L1 data memory.

Data Memory Control Register (DMEM_CONTROL)

The data memory control register (`DMEM_CONTROL`) contains control bits for the L1 data memory. See Figure 3-9 on page 3-28.

The `PORT_PREF1` bit selects the data port used to process DAG1 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to memory other than L1 full.

The `PORT_PREF0` bit selects the data port used to process DAG0 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to memory other than L1 full.

Data Memory Control Register (DMEM_CONTROL)

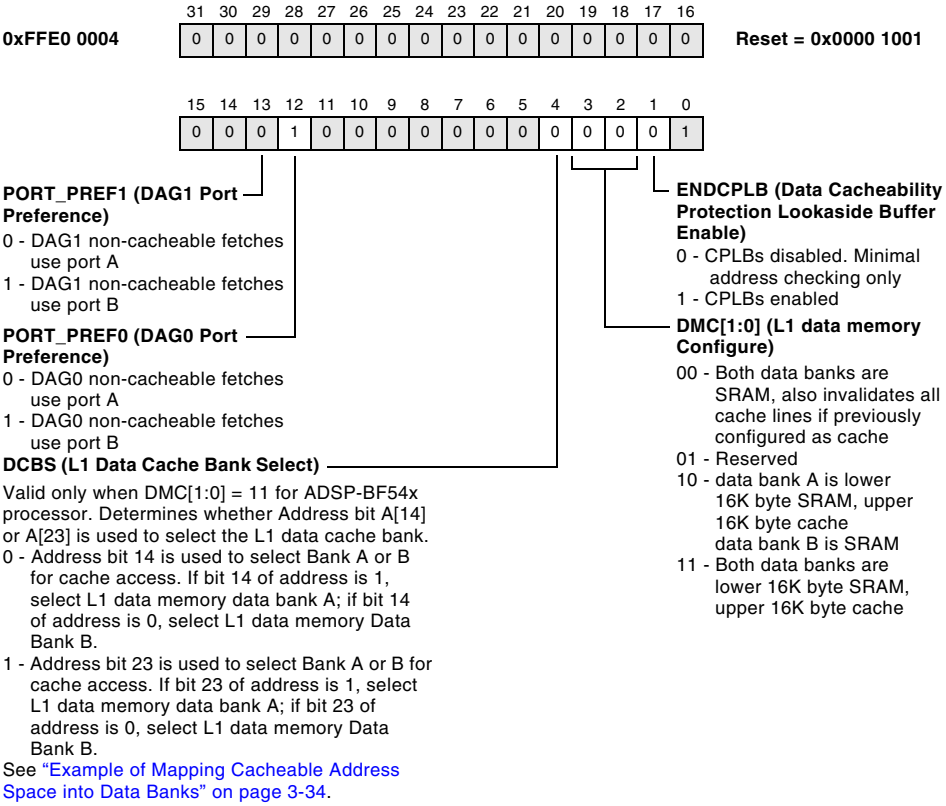


Figure 3-9. L1 Data Memory Control Register

For optimal performance with dual DAG reads, DAG0 and DAG1 should be configured for different ports. For example, if `PORT_PREF0` is configured as 1, then `PORT_PREF1` should be programmed to 0.

The DCBS bit provides some control over which addresses alias into the same set. This bit can be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets. It has no affect unless both data bank A and data bank B are serving as cache (bits DMC[1:0] in this register are set to 11).

The ENDCPLB bit is used to enable/disable the 16 cacheability protection lookaside buffers (CPLBs) used for data (see "L1 Data Cache" on page 3-33). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in user mode
- Attempts to write the on-chip boot ROM

CPLBs must be disabled using this bit prior to updating their descriptors (registers DCPLB_DATAx and DCPLB_ADDRx). Note that since load store ordering is weak (see "Ordering of Loads and Stores" on page 3-72), disabling CPLBs should be preceded by a CSYNC instruction, and enabling CPLBs should be followed by a CSYNC instruction in order to ensure predictable behavior.

When enabling or disabling cache or CPLBs, immediately follow the write to DMEM_CONTROL with a SSYNC to ensure proper behavior.

By default after reset, all L1 data memory serves as SRAM. The DMC[1:0] bits can be used to reserve portions of this memory to serve as cache instead. Reserving memory to serve as cache does not enable memory other than L1 accesses to be cached. To do this, CPLBs must also be enabled (using the ENDCPLB bit) and CPLB descriptors (registers DCPLB_DATAx and DCPLB_ADDRx) must specify chosen memory pages as cache-enabled.

By default after reset, cache and CPLB address checking is disabled.

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

L1 Data SRAM

Accesses to SRAM do not collide unless they are to the same 32-bit word polarity (address bits 2 match), the same 4K byte sub-bank (address bits 13 and 12 match), the same 16K byte half-bank (address bits 16 match), and the same bank (address bits 21 and 20 match). When an address collision is detected, access is nominally granted first to the DAGs, then to the store buffer, and finally to the DMA and cache fill/victim traffic. To ensure adequate DMA bandwidth, DMA is given highest priority if it is blocked for more than 16 sequential core clock cycles, or if a second DMA I/O is queued before the first DMA I/O is processed.

Table 3-4 shows how the subbank organization is mapped into memory.

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

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

Figure 3-10 on page 3-32 shows the L1 data memory architecture.

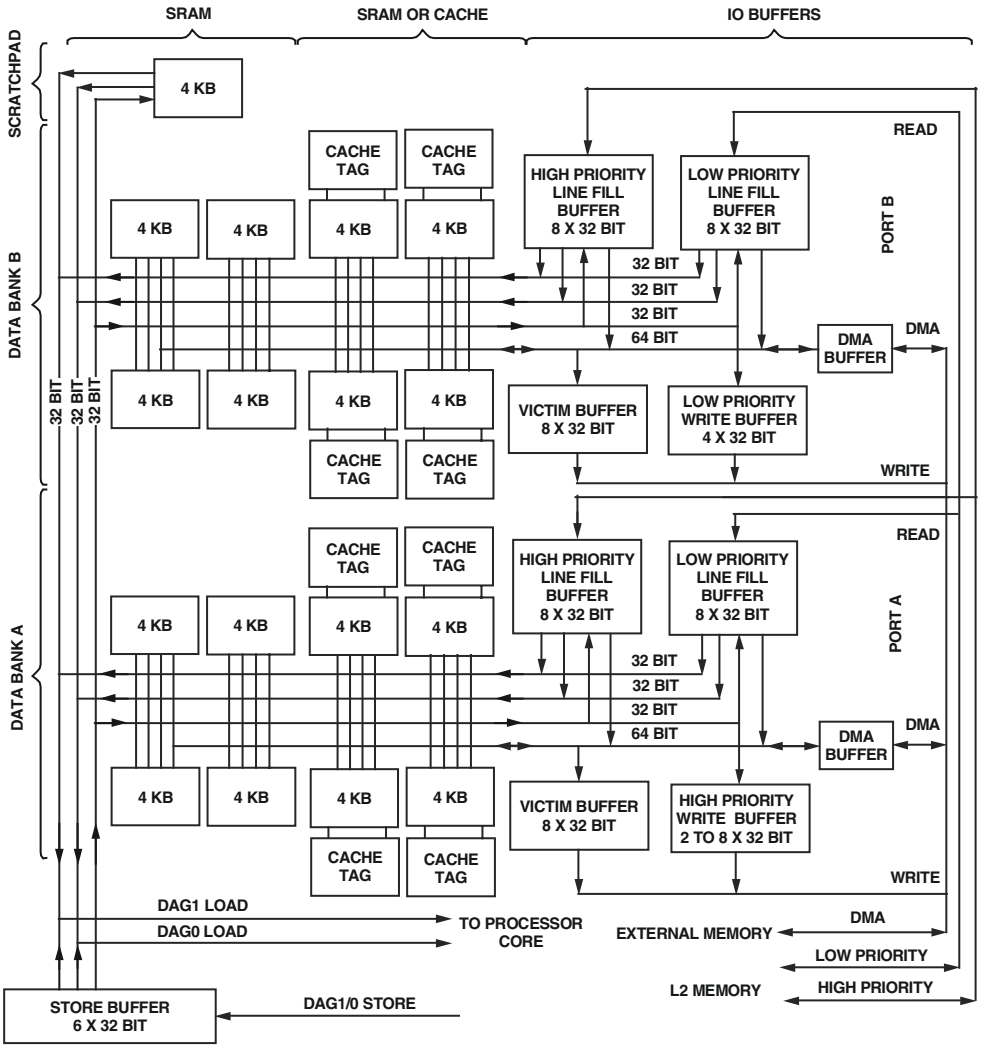


Figure 3-10. L1 Data Memory Architecture

L1 Data Cache

For definitions of cache terminology, see "Terminology" on page 3-79.

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-BF54x processor processor, the upper 16K byte is used. Unlike instruction cache, which is 4-way set associative, data cache is 2-way set-associative. When two banks are available and enabled as cache, additional sets rather than ways are created. When both data bank A and data bank B have memory serving as cache, the DCBS bit in the DMEM_CONTROL register may be used to control which half of all address space is handled by which bank of cache memory. The DCBS bit selects either address bit 14 or 23 to steer traffic between the cache banks. This provides some control over which addresses alias into the same set. It may therefore be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Accesses to cache do not collide unless they are to the same 4K byte sub-bank (address bits 13 and 12 match), the same half bank (address bits 16 match), and to the same bank (address bits 21 and 20 match). Cache has less apparent multiported behavior due to the overhead in maintaining tags. When cache addresses collide, access is granted first to the DTEST register accesses, then to the store buffer, and finally to cache fill/victim traffic.

Three different cache modes are available:

- Write-through with cache line allocation only on reads
- Write-through with cache line allocation on both reads and writes
- Write-back which allocates cache lines on both reads and writes

Cache mode is selected by the DCPLB descriptors (see "Memory Protection and Properties" on page 3-51). Any combination of these cache modes can be used simultaneously since cache mode is selectable for each memory page independently.

If cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), data CPLBs should also be enabled (controlled by ENDCPLB bit in the DMEM_CONTROL register). Only memory pages specified as cacheable by data CPLBs are cached. The default behavior is no caching when data CPLBs are disabled.

Erroneous behavior can result when MMR space is configured as cacheable by data CPLBs, or when data banks serving as L1 SRAM are configured as cacheable by data CPLBs.

Example of Mapping Cacheable Address Space into Data Banks

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

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

If both data banks are configured as cache, the DCBS bit in the DMEM_CONTROL register designates address bit A[14] or A[23] as the cache selector. Address bit A[14] or A[23] selects the cache implemented by data bank A or the cache implemented by data bank B.

- If DCBS = 0, then A[14] is part of the address index, and all addresses in which A[14] = 0 use data bank A. All addresses in which A[14] = 1 use data bank B.
- In this case, A[23] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.
- If DCBS = 1, then A[23] is part of the address index, and all addresses where A[23] = 0 use data bank A. All addresses where A[23] = 1 use data bank B.
- In this case, A[14] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing DCBS = 0 or DCBS = 1 is:

- If DCBS = 0, A[14] selects data bank A instead of data bank B.
- Alternating 16K byte pages of memory map into each of the two 16K byte caches implemented by the two data banks. Consequently:
    - Any data in the first 16K byte of memory could be stored only in data bank A.
    - Any data in the next address range (16K byte through 32K byte) – 1 could be stored only in data bank B.
    - Any data in the next range (32K byte through 48K byte) – 1 would be stored in data bank A.
    - Alternate mapping would continue.

- As a result, the cache operates as if it were a single, contiguous, 2-way set associative 32K byte cache. Each way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each way).
- If DCBS = 1, A[23] selects data bank A instead of data bank B.
- With DCBS = 1, the system functions more like two independent caches, each a 2-way set associative 16K byte cache. Each Bank serves an alternating set of 8M byte blocks of memory. For example, data bank A caches all data accesses for the first 8M byte of memory address range. That is, every 8M byte of range vies for the two line entries (rather than every 16K byte repeat). Likewise, data bank B caches data located above 8M byte and below 16M byte.
- For example, if the application is working from a data set that is 1 Mbyte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by data bank A (a 2-way set associative 16K byte cache). In this instance, the application never derives any benefit from data bank B.

For most applications, it is best to operate with DCBS = 0.

However, if the application is working from two data sets, located in two memory spaces at least 8 Mbyte apart, closer control over how the cache maps to the data is possible. For example, if the program is doing a series of dual MAC operations in which both DAGs are accessing data on every cycle, by placing DAG0's data set in one block of memory and DAG1's data set in the other, the system can ensure that:

- DAG0 gets its data from data bank A for all of its accesses,
- DAG1 gets its data from data bank B.

This arrangement causes the core to use both data buses for cache line transfer and achieves the maximum data bandwidth between the cache and the core.

Figure 3-11 shows an example of how mapping is performed when `DCBS = 1`.

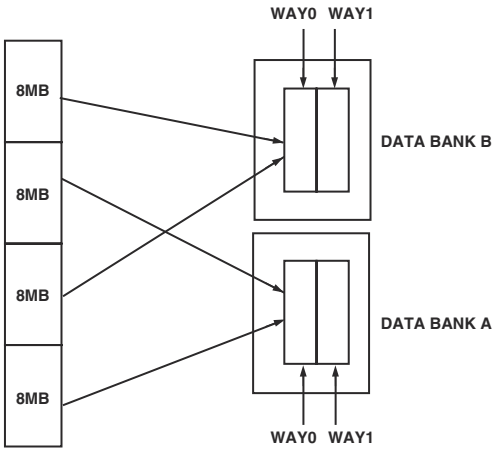


Figure 3-11. Data Cache Mapping When DCBS = 1

The `DCBS` selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.

Data Cache Access

The cache controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory through the system interface. The line index and replacement policy for the cache controller determines the cache tag and data space that are allocated for the data coming back from memory other than L1.

A data cache line is in one of three states: invalid, exclusive (valid and clean), and modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

- If the state of the line is exclusive (clean), the new tag and data write over the old line.
- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data.
- If the line is dirty, the current contents of the cache are copied back to memory other than L1 before new data is written to the cache.

The processor provides victim buffers and line fill buffers. These buffers are used if a cache load miss generates a victim cache line that should be replaced. The line fill operation goes to memory other than L1. The data cache performs the line fill request to the system as critical (or requested) word first, and forwards that data to the waiting DAG as it updates the cache line. In other words, the cache performs critical word forwarding.

The data cache supports hit-under-a-store miss, and hit-under-a-prefetch miss. In other words, on a write-miss or execution of a `PREFETCH` instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a four-cycle stall. Furthermore, a subsequent load or store instruction can hit in the L1 cache while the line fill completes.

Interrupts of sufficient priority (relative to the current context) cancel a stalled load instruction. Consequently, if the load operation misses the L1 data memory cache and generates a high-latency line fill operation on the system interface, it is possible to interrupt the core, causing it to begin processing a different context. The system access to fill the cache line is not cancelled, and the data cache is updated with the new data before any further cache miss operations to the respective data bank are serviced. For more information see "System Interrupts" on page 6-1.

Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to memory other than L1 immediately upon the write to cache.
- If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to memory other than L1.
- A write-back cache does not write to memory other than L1 until the line is replaced by a load operation that needs the line.

  The L1 data memory employs a full-cache, line-width copyback buffer on each data bank.

Write Buffers

Two separate write buffers are provided. These buffers allow stores to slow external memory to continue without causing stores to higher-speed on-chip memory other than L1 to stall. Which buffer is used is determined by the `CPLB_MEMLEV` bit in the data memory page's CPLBs. See "Memory Protection and Properties" on page 3-51.

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

An `SSYNC` instruction flushes the write buffers.

Interrupt Priority Register (IPRIO) and Write Buffer Depth

The interrupt priority register (IPRIO) can be used to control the size of the high priority write buffer on port A (see Figure 3-10 on page 3-32).

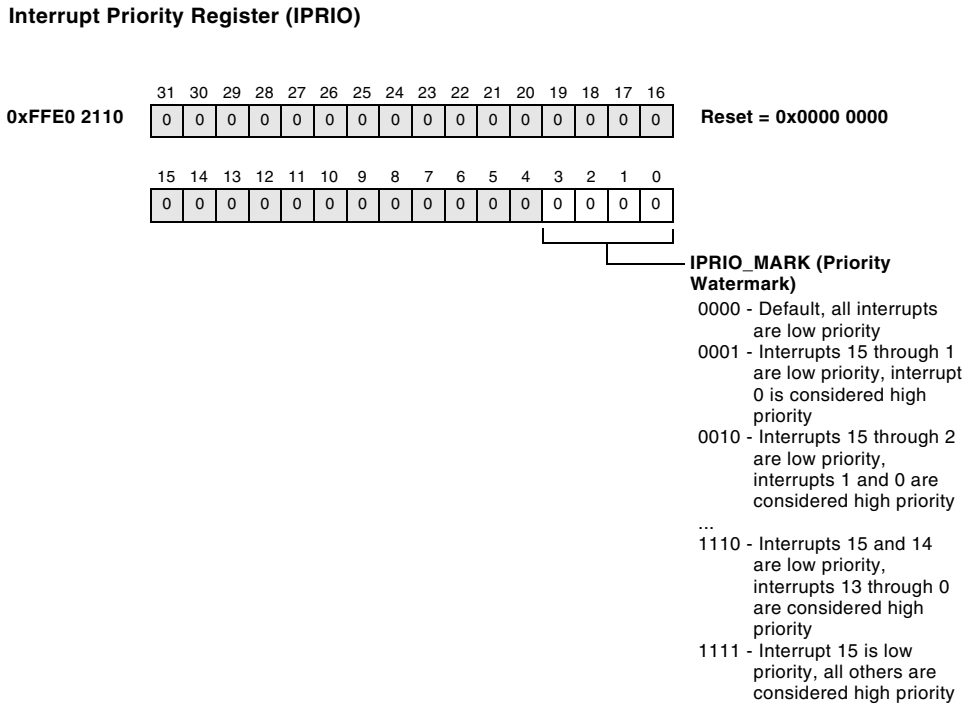


Figure 3-12. Interrupt Priority Register

The IPRIO[3:0] bits can be programmed to reflect the low priority interrupt watermark. When an interrupt occurs, causing the processor to vector from a low priority interrupt service routine to a high priority interrupt service routine, the size of the low priority write buffer increases from two to eight 32-bit words deep. This allows the interrupt service routine to run and post writes without an initial stall, in the case where the low priority write buffer was already filled in the low priority interrupt routine. This is most useful when posted writes are to a slow external memory

device. When returning from a high priority interrupt service routine to a low priority interrupt service routine or user mode, the core stalls until the write buffer has completed the necessary writes to return to a two-deep state. By default, the low priority write buffer is a fixed two-deep FIFO.

Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in user and supervisor modes. The instructions are `PREFETCH`, `FLUSH`, and `FLUSHINV`.

- `PREFETCH` (data cache prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache inhibited region, `PREFETCH` functions like a `NOP`.
- `FLUSH` (data cache flush) causes the data cache to synchronize the specified cache line with memory other than L1. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, `FLUSH` functions like a `NOP`.
- `FLUSHINV` (data cache line flush and invalidate) causes the data cache to perform the same function as the `FLUSH` instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to memory other than L1. The valid bit in the cache line is then cleared. If the line is not in the cache, `FLUSHINV` functions like a `NOP`.

If software requires synchronization with system hardware, place an `SSYNC` instruction after the `FLUSH` instruction to ensure that the flush operation has completed. If ordering is desired to ensure that previous stores have been pushed through all the queues, place an `SSYNC` instruction before the `FLUSH`.

Data Cache Invalidation

Besides the FLUSHINV instruction, two additional methods are available to invalidate the data cache when flushing is not required. The first technique directly invalidates valid bits by setting the Invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (DTEST_COMMAND and DTEST_DATA[1:0]) are available to allow arbitrary reads/writes of all the cache entries directly.

For invalidating the complete data cache, a second method is available. By clearing the DMC[1:0] bits in the DMEM_CONTROL register (see Figure 3-9 on page 3-28), all valid bits in the data cache are set to the invalid state. A second write to the DMEM_CONTROL register sets the DMC[1:0] bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Data Test Registers

Like L1 instruction memory, L1 data memory contains additional MMRs to allow arbitrary reads/writes of all cache entries directly. The registers provide a mechanism for data cache test, initialization, and debug.

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed and data is transferred through the data test data registers (DTEST_DATA[1:0]). The DTEST_DATA[1:0] registers contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The lower 32 bits are stored in the DTEST_DATA[0] register and the upper 32 bits are stored in the DTEST_DATA[1] register. When the tag arrays are accessed, the DTEST_DATA[0] register is used.

A CSYNC instruction is required after writing the DTEST_COMMAND MMR.

The DTEST registers are described in the following subsections.

Table 3-5. DTEST Registers

Name	Description/ Refer to
DTEST_COMMAND	Data test command register For more information, see "Data Test Command Register (DTEST_COMMAND)" on page 3-44.
DTEST_DATA1	Data test data 1 register For more information, see "Data Test Data 1 Register (DTEST_DATA1)" on page 3-45.
DTEST_DATA0	Data test data 0 register For more information, see "Data Test Data 0 Register (DTEST_DATA0)" on page 3-46.

Access to these registers is possible only in supervisor or emulation mode. When writing to DTEST registers, always write to the DTEST_DATA registers first, then the DTEST_COMMAND register.

Data Test Command Register (DTEST_COMMAND)

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

The data/instruction access bit allows direct access by way of the DTEST_COMMAND MMR to L1 instruction SRAM. Note that L1 instruction ROM is not directly accessible. Instruction ROM is accessible only through instruction fetches or DMA accesses.

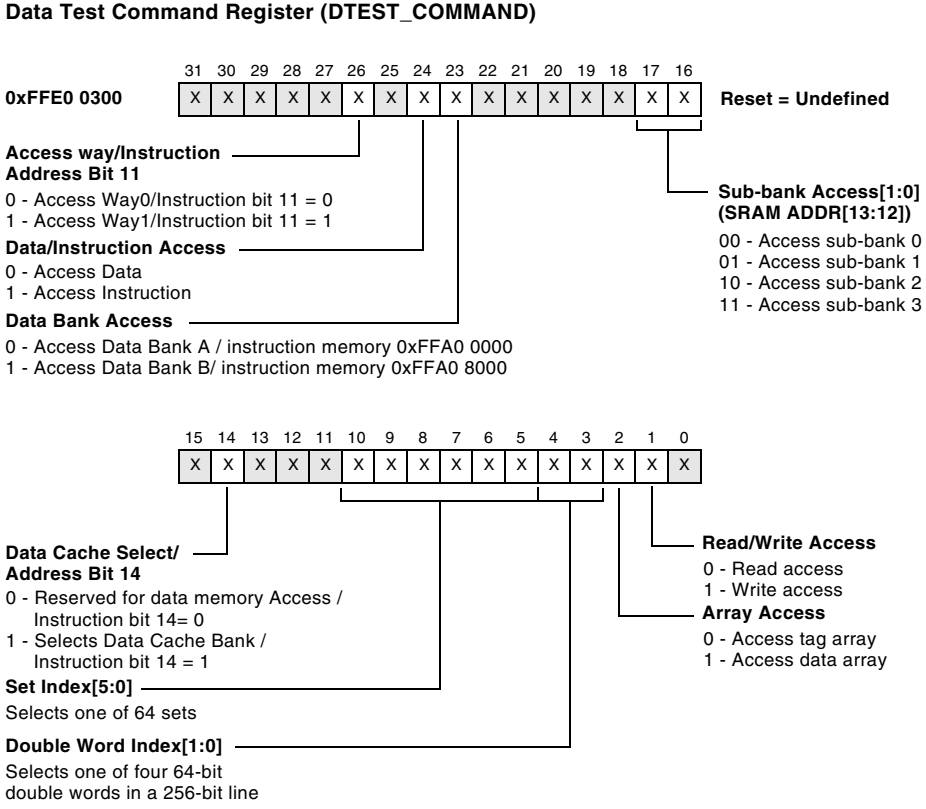


Figure 3-13. Data Test Command Register

Data Test Data 1 Register (DTEST_DATA1)

Data test data registers (DTEST_DATA[1:0]) contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The data test data 1 register (DTEST_DATA1) stores the upper 32 bits.

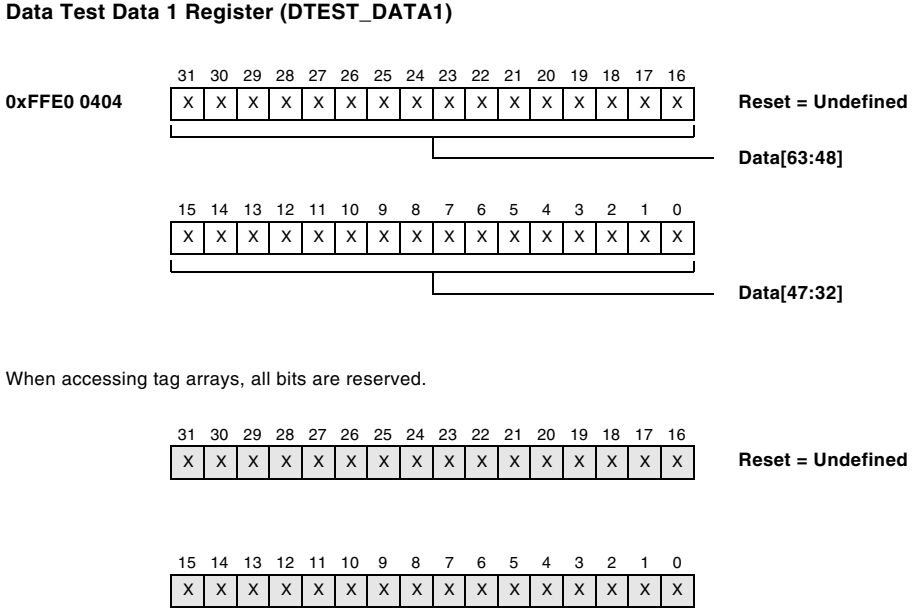


Figure 3-14. Data Test Data 1 Register

Data Test Data 0 Register (DTEST_DATA0)

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

The DTEST_DATA0 register is also used to access the tag arrays and contains the valid and dirty bits, which indicate the state of the cache line.

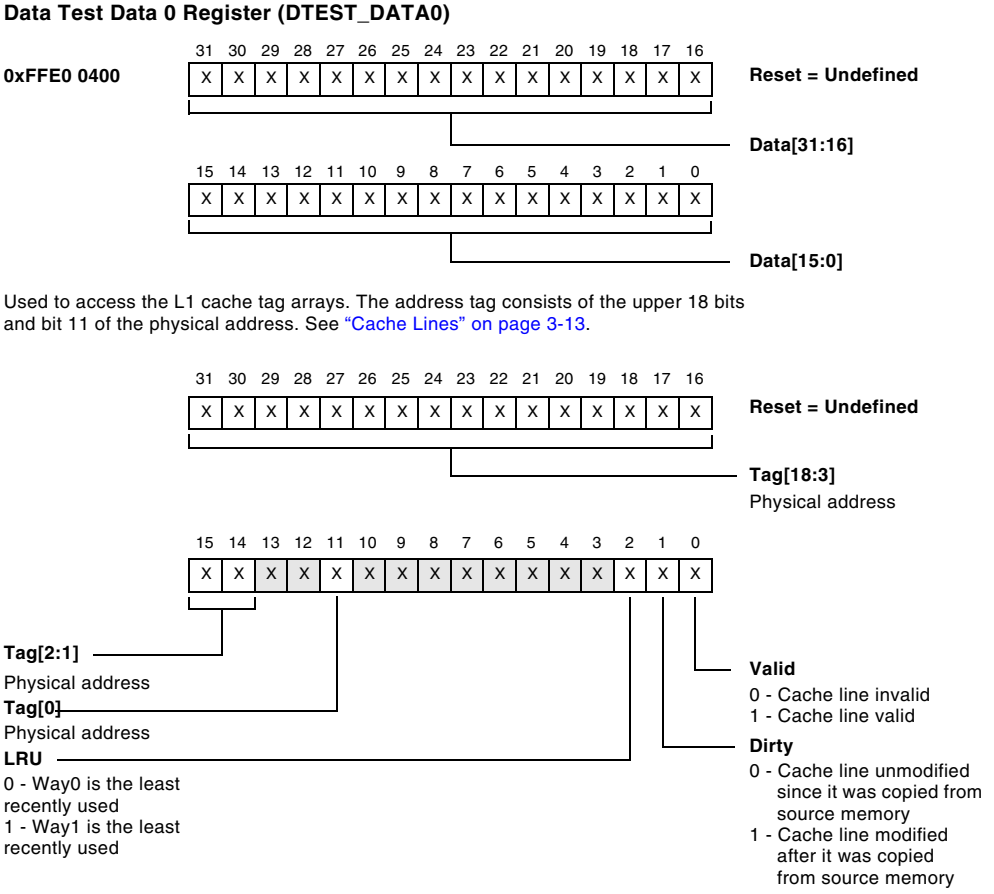


Figure 3-15. Data Test Data 0 Register

On-Chip Level 2 (L2) Memory

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

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

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

On-Chip L2 Bank Access

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

Latency

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

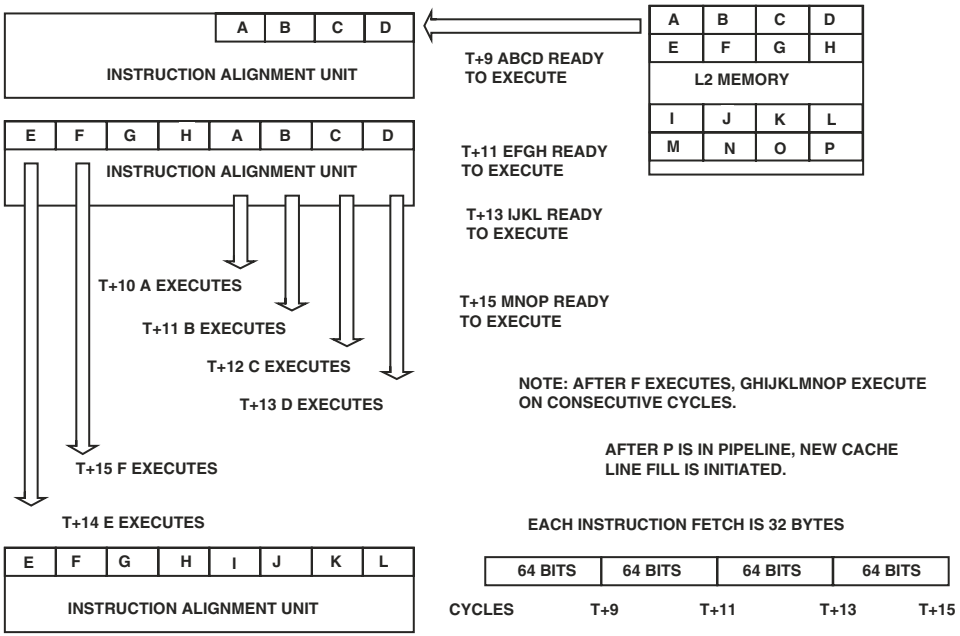


Figure 3-16. L2 Latency With Cache On

In this example, at the end of 15 core cycles, 32 bytes of instructions or data have been brought into cache and are available to the sequencer. If all the instructions contain 16 bits, sixteen instructions are brought into cache at the end of 15 cycles. In addition, the first instruction that is part of the cache line fill executes on the tenth cycle; the second instruction executes on the eleventh cycle, and the third instruction executes on the twelfth cycle—all of them in parallel with the cache line fill.

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

When on-chip L2 is configured as non-cacheable, instruction fetches and data fetches occur in 64-bit fills. In this case, each fill takes seven core cycles to complete. As shown in Figure 3-17 on page 3-50, on-chip L2 is configured as non-cacheable. To illustrate the concept of L2 latency with cache off, simple instructions are used that do not require additional external data fetches. In this case, consecutive instructions are issued on consecutive cycles if multiple instructions are brought into the core in a given fetch.

One Time Programmable Memory

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

Because OTP memory usage is required for usage of the security features of the ADSP-BF54x processor processor, OTP memory is described in Chapter 4, One-Time Programmable Memory. Note that OTP memory has many other uses besides support for security.

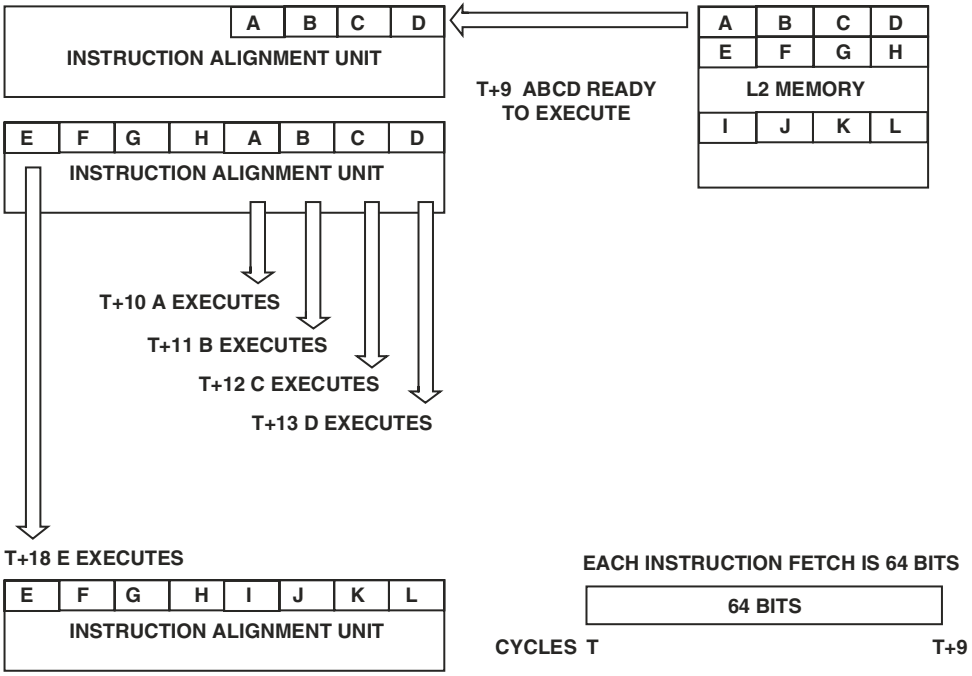


Figure 3-17. L2 Latency With Cache Off

External Memory

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

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

Memory Protection and Properties

This section describes the memory management unit (MMU), memory pages, CPLB management, MMU management, and CPLB registers.

Memory Management Unit

The Blackfin processor contains a page-based memory management unit (MMU). This mechanism provides control over cacheability of memory ranges, as well as management of protection attributes at page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.

The MMU is implemented as two 16-entry content addressable memory (CAM) blocks. Each entry is referred to as a cacheability protection lookaside buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction.

Because L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries are used for instruction fetch requests; these are called ICPLBs. Another sixteen CPLB entries are used for data transactions; these are called DCPLBs. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 instruction memory Control

(IMEM_CONTROL) and L1 data memory control (DMEM_CONTROL) registers, respectively. These registers are shown in Figure 3-2 on page 3-9 and Figure 3-9 on page 3-28.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetches:

- ICPLB_ADDR[n] defines the start address of the page described by the CPLB descriptor.
- ICPLB_DATA[n] defines the properties of the page described by the CPLB descriptor.

For data operations:

- DCPLB_ADDR[m] defines the start address of the page described by the CPLB descriptor.
- DCPLB_DATA[m] defines the properties of the page described by the CPLB descriptor.

There are two default CPLB descriptors for data accesses to the scratchpad data memory and to the system and core MMR space. These default descriptors define the above space as non-cacheable, so that additional CPLBs do not need to be set up for these regions of memory.

If valid CPLBs are set up for this space, the default CPLBs are ignored.

Memory Pages

The 4G byte address space of the processor can be divided into smaller ranges of memory or I/O referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports four different page sizes:

- 1K byte
- 4K byte
- 1M byte
- 4M byte

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

Memory Page Attributes

Each page is defined by a two-word descriptor, consisting of an address descriptor word `xCPLB_ADDR[n]` and a properties descriptor word `xCPLB_DATA[n]`. The address descriptor word provides the base address of the page in memory. Pages must be aligned on page boundaries that are an integer multiple of their size. For example, a 4M byte page must start on an address divisible by 4M byte; whereas a 1K byte page can start on any 1K byte boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size
- 1K byte, 4K byte, 1M byte, 4M byte
- Cacheable/non-cacheable
  Accesses to this page use the L1 cache or bypass the cache.
  If cacheable: write-through/write-back data writes propagate directly to memory or are deferred until the cache line is reallocated. If write-through, allocate on read-only, or read and write.

- Dirty/modified
  The data memory in this page has changed since the CPLB was last loaded.
- Supervisor write access permission
  Enables or disables writes to this page when in supervisor mode. Data pages only.
- User write access permission
  Enables or disables writes to this page when in user mode. Data pages only
- User read access permission
  Enables or disables reads from this page when in user mode
- Valid
  Check this bit to determine whether this is valid CPLB data
- Lock
  Keep this entry in MMR; do not participate in CPLB replacement policy.

Page Descriptor Table

For memory accesses to utilize the cache when CPLBs are enabled for instruction access, data access, or both, a valid CPLB entry must be available in an MMR pair. The MMR storage locations for CPLB entries are limited to 16 descriptors for instruction fetches and 16 descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB descriptors that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a *static* memory management model.

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than can fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a page descriptor table, is used; in it can be stored all the potentially required CPLB descriptors. The specific format for the page descriptor table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement page descriptor table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

CPLB Management

When the Blackfin processor issues a memory operation for which no valid CPLB (cacheability protection look aside buffer) descriptor exists in an MMR pair, an exception occurs that places the processor into supervisor mode and vectors to the MMU exception handler (see "System Interrupts" on page 6-1 for more information). The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

Before CPLBs are enabled, valid CPLB descriptors must be in place for both the page descriptor table and the MMU exception handler. The `LOCK` bits of these CPLB descriptors are commonly set so they are not inadvertently replaced in software.

The handler uses the faulting address to index into the page descriptor table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of 16 CPLBs must be disabled using:

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

The CPLB replacement policy and algorithm used are the responsibility of the system MMU exception handler. This policy, which is dictated by the characteristics of the operating system, usually implements a modified LRU (least recently used) policy, a round-robin scheduling method, or pseudo random replacement.

After the new CPLB descriptor is loaded, the exception handler returns, and the faulting memory operation is restarted. This operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.

A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid CPLB descriptor in an MMR pair. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- A page miss on DAG0
- A page miss on DAG1

MMU Application

Memory management is an optional feature in the Blackfin processor architecture. Its use is predicated on the system requirements of a given application. Upon reset, all CPLBs are disabled, and the memory management unit (MMU) is not used.

If all L1 memory is configured as SRAM, then the data and instruction MMU functions are optional, depending on the application's need for protection of memory spaces either between tasks or between user and supervisor modes. To protect memory between tasks, the operating system can maintain separate tables of instruction and/or data memory pages available for each task and make those pages visible only when the relevant task is running. When a task switch occurs, the operating system can ensure the invalidation of any CPLB descriptors on chip that should not be available to the new task. It can also preload descriptors appropriate to the new task.

For many operating systems, the application program is run in user mode while the operating system and its services run in supervisor mode. It is desirable to protect code and data structures used by the operating system from inadvertent modification by a running user mode application. This protection can be achieved by defining CPLB descriptors for protected memory ranges that allow write access only when in supervisor mode. If a write to a protected memory region is attempted while in user mode, an exception is generated before the memory is modified. Optionally, the user mode application may be granted read access for data structures that are useful to the application. Even supervisor mode functions can be blocked from writing some memory pages that contain code that is not expected to be modified. Because CPLB entries are MMRs that can be written only while in supervisor mode, user programs cannot gain access to resources protected in this way.

If either the L1 instruction memory or the L1 data memory is configured partially or entirely as cache, the corresponding CPLBs must be enabled. When an instruction generates a memory request and the cache is enabled, the processor first checks the ICPLBs to determine whether the address requested is in a cacheable address range. If no valid ICPLB entry in an MMR pair corresponds to the requested address, an MMU exception is generated to obtain a valid ICPLB descriptor to determine whether the memory is cacheable or not. As a result, if the L1 instruction memory is enabled as cache, then any memory region that contains instructions must

have a valid ICPLB descriptor defined for it. These descriptors must either reside in MMRs at all times or be resident in a memory-based page descriptor table that is managed by the MMU exception handler. Likewise, if either or both L1 data banks are configured as cache, all potential data memory ranges must be supported by DCPLB descriptors.

Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.

Examples of Protected Memory Regions

In Figure 3-18, a starting point is provided for basic CPLB allocation for instruction and data CPLBs. Note some ICPLBs and DCPLBs have common descriptors for the same address space.

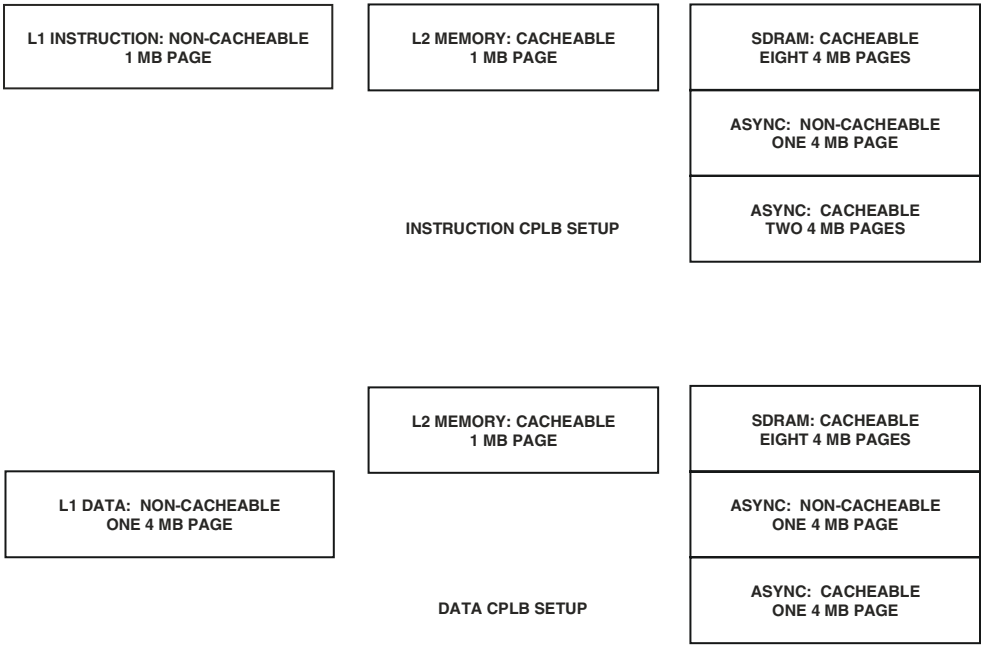


Figure 3-18. Examples of Protected Memory Regions

ICPLB Data Registers (ICPLB_DATAx)

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

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

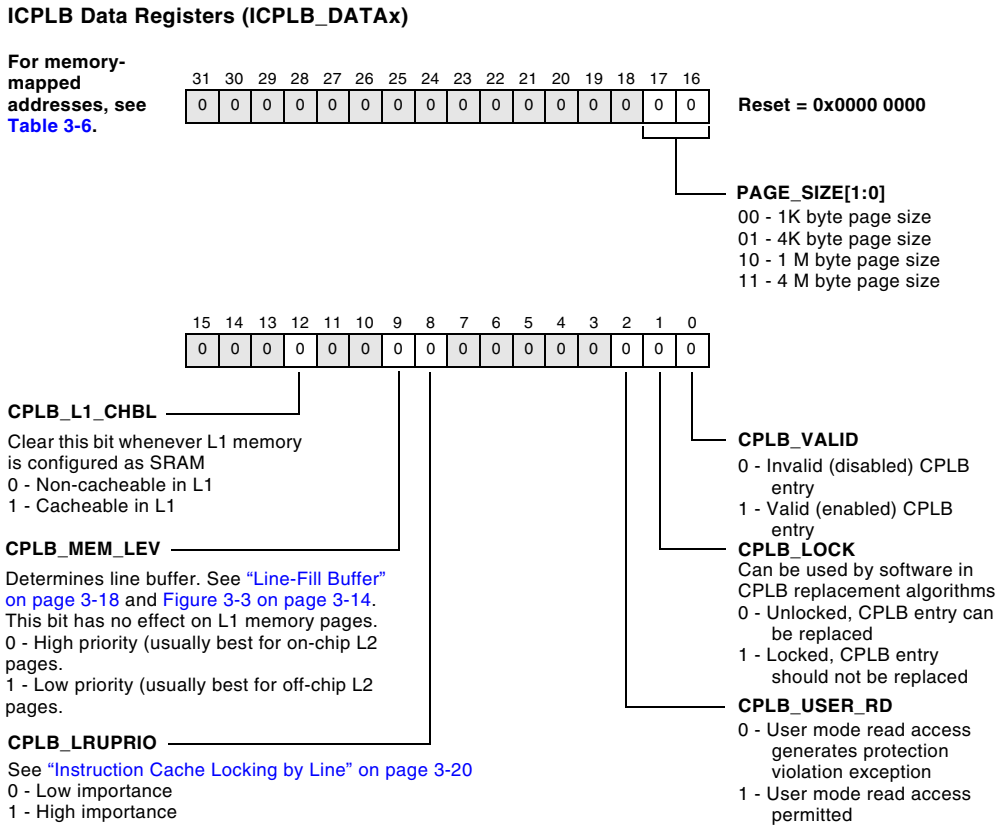


Figure 3-19. ICPLB Data Registers

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

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

DCPLB Data Registers (DCPLB_DATAx)

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

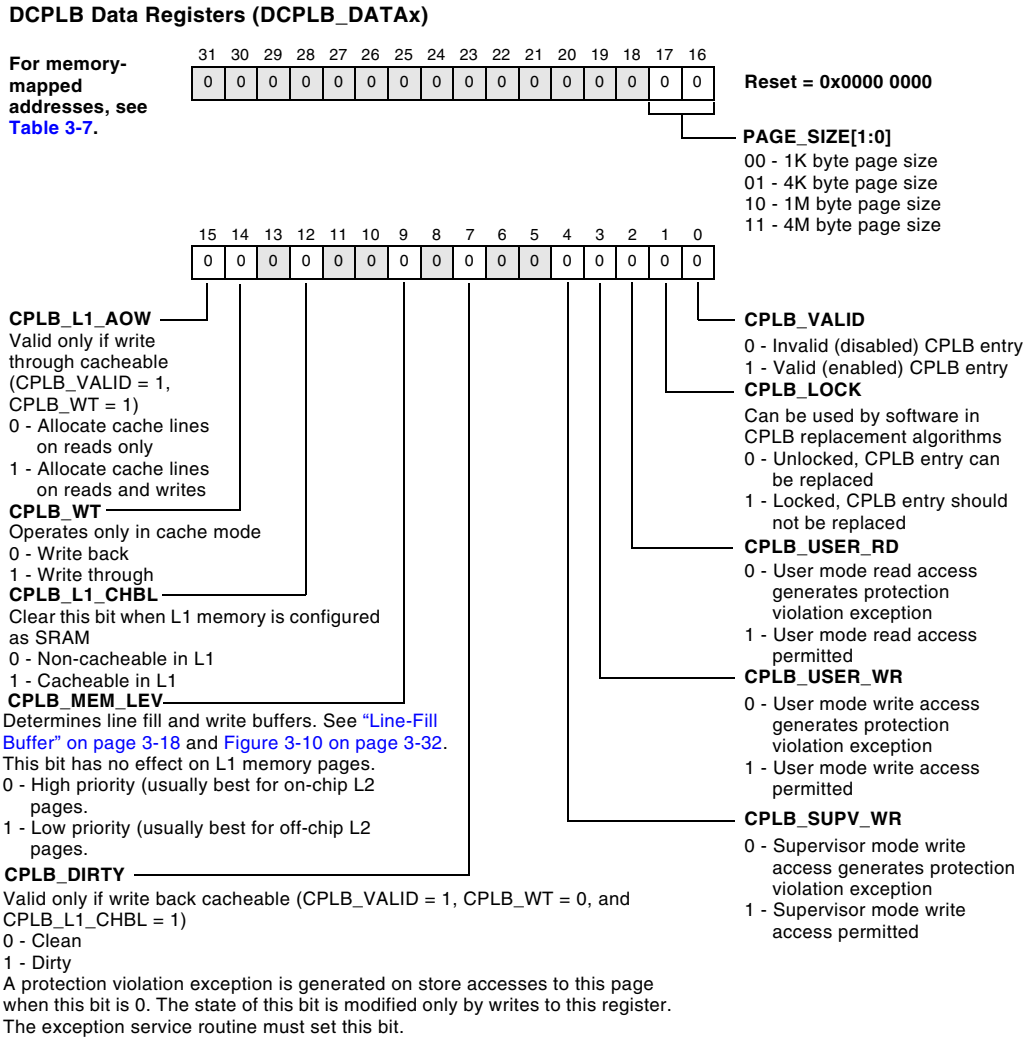


Figure 3-20. DCPLB Data Registers

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

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

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

DCPLB Address Registers (DCPLB_ADDRx)

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

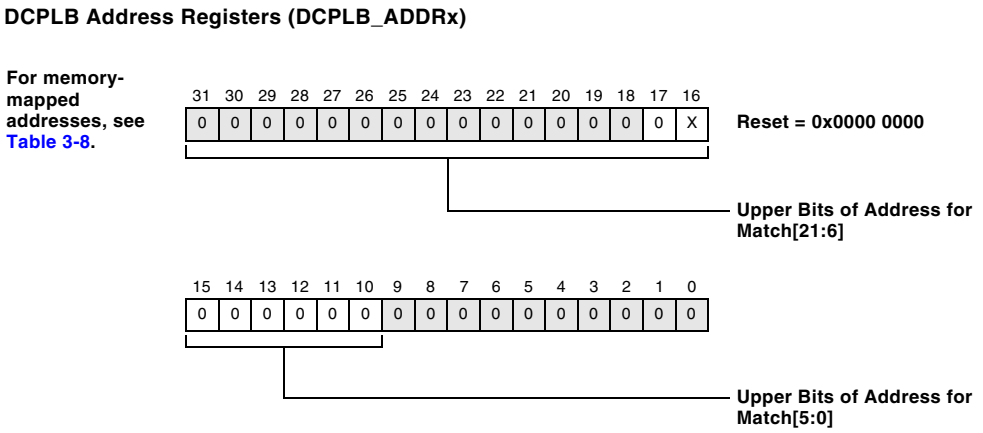


Figure 3-21. DCPLB Address Registers

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

Register Name	Memory-Mapped Address
DCPLB_ADDR0	0xFFE0 0100
DCPLB_ADDR1	0xFFE0 0104
DCPLB_ADDR2	0xFFE0 0108
DCPLB_ADDR3	0xFFE0 010C
DCPLB_ADDR4	0xFFE0 0110
DCPLB_ADDR5	0xFFE0 0114
DCPLB_ADDR6	0xFFE0 0118
DCPLB_ADDR7	0xFFE0 011C
DCPLB_ADDR8	0xFFE0 0120
DCPLB_ADDR9	0xFFE0 0124

Table 3-8. DCPLB Address Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DCPLB_ADDR10	0xFFE0 0128
DCPLB_ADDR11	0xFFE0 012C
DCPLB_ADDR12	0xFFE0 0130
DCPLB_ADDR13	0xFFE0 0134
DCPLB_ADDR14	0xFFE0 0138
DCPLB_ADDR15	0xFFE0 013C

ICPLB Address Registers (ICPLB_ADDRx)

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

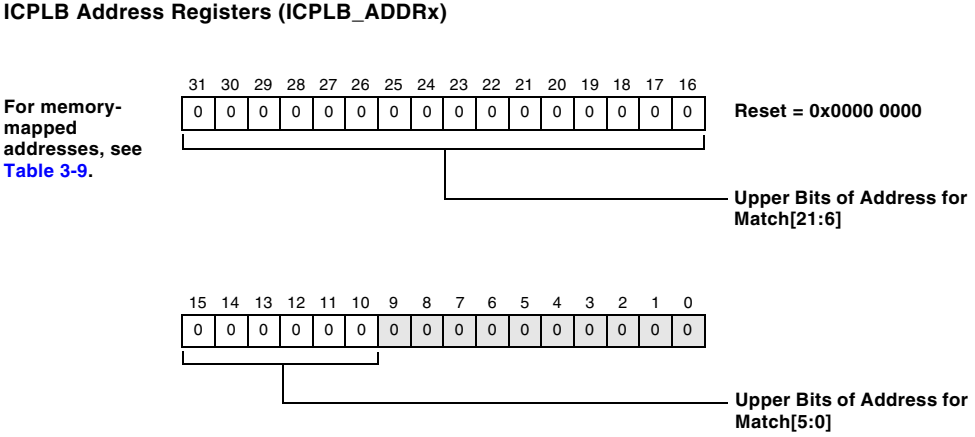


Figure 3-22. ICPLB Address Registers

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

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

CPLB Status Registers

Bits in the DCPLB status register (`DCPLB_STATUS`) and ICPLB status register (`ICPLB_STATUS`) identify the CPLB entry that triggered CPLB-related exceptions. The exception service routine can infer the cause of the fault by examining the CPLB entries.

The `DCPLB_STATUS` and `ICPLB_STATUS` registers are valid only while in the faulting exception service routine.

DCPLB Status Register (DCPLB_STATUS)

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

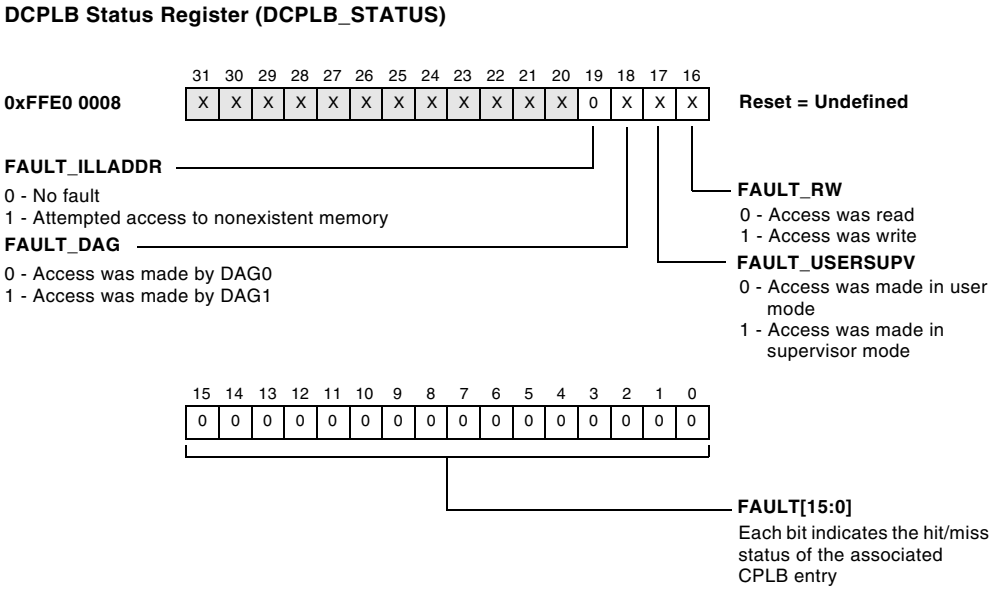


Figure 3-23. DCPLB Status Register

ICPLB Status Register (ICPLB_STATUS)

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

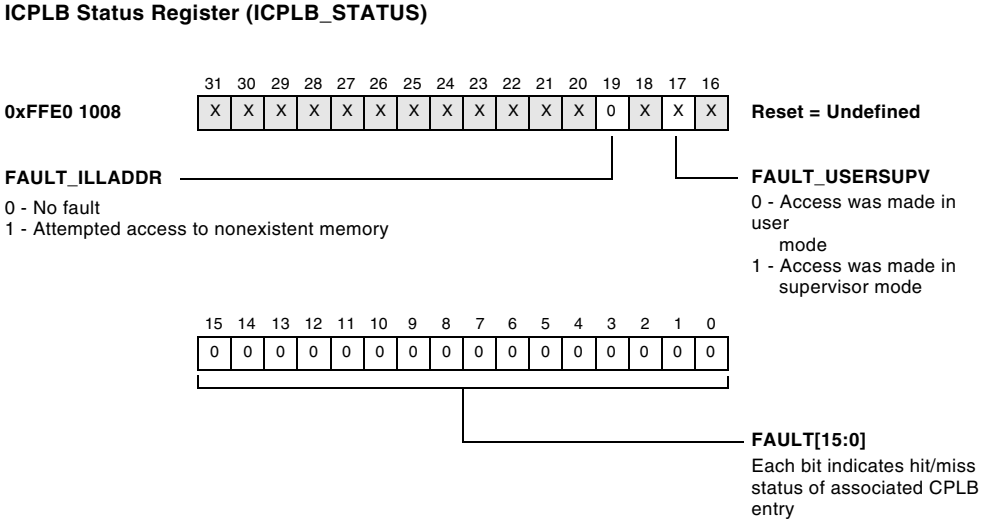


Figure 3-24. ICPLB Status Register

CPLB Fault Address Registers

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

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

DCPLB Fault Address Register (DCPLB_FAULT_ADDR)

Figure 3-25 lists the DCPLB fault address register.

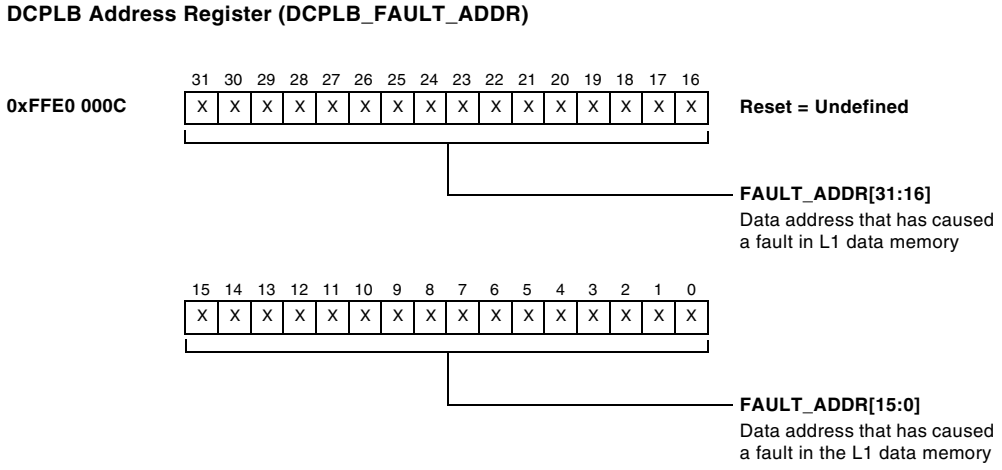


Figure 3-25. DCPLB Fault Address Register

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

Figure 3-26 lists the ICPLB fault address register.

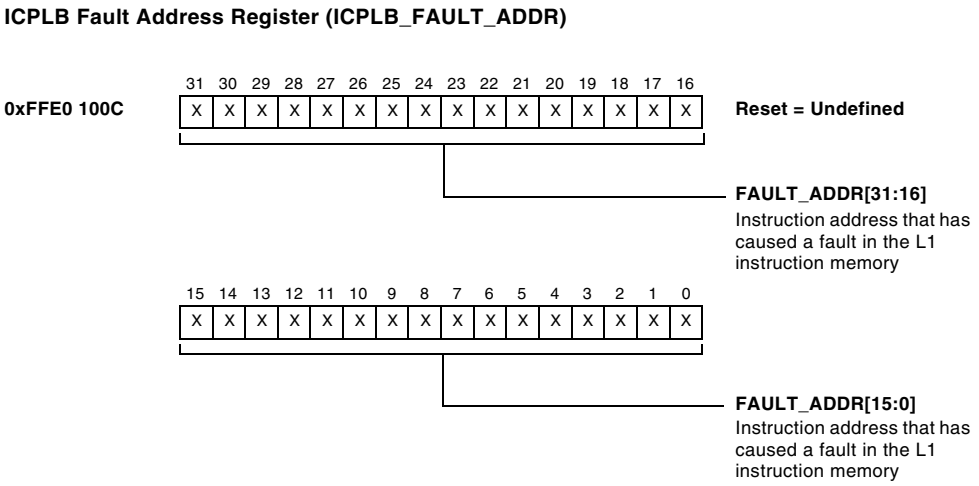


Figure 3-26. ICPLB Fault Address Register

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 3-27 shows a data word stored in register `R0` and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

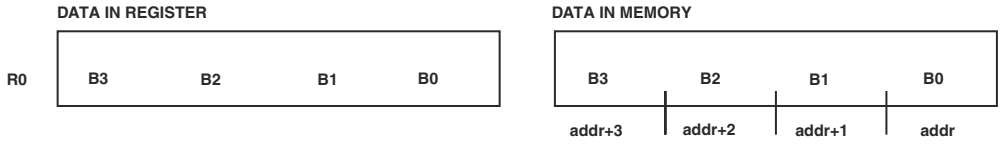


Figure 3-27. Data Stored in Little Endian Order

Figure 3-28 shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in *addr+1*) and the least significant byte in the low address (byte B0 in *addr*).

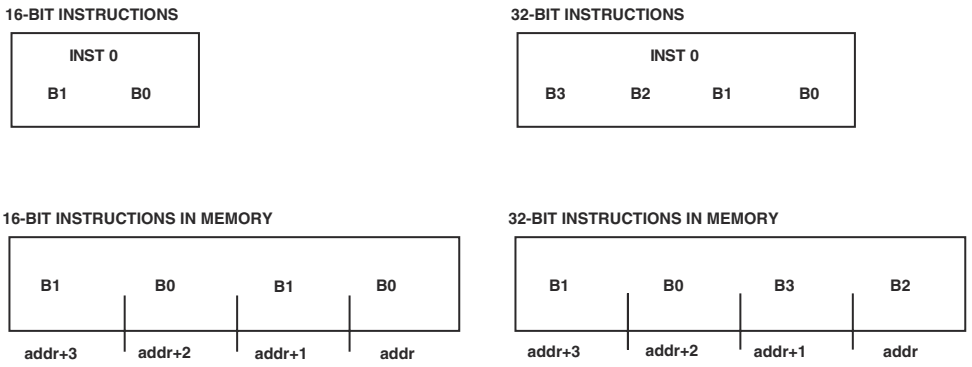


Figure 3-28. Instructions Stored in Little Endian Order

The diagram on the right shows 32-bit instructions stored in memory. Note the most significant 16-bit half word of the instruction (bytes B3 and B2) is stored in the low addresses (*addr+1* and *addr*), and the least significant half word (bytes B1 and B0) is stored in the high addresses (*addr+3* and *addr+2*).

Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a load/store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particu-

larly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

Separating load operations from their associated arithmetic functions allows compilers or assembly language programmers to place unrelated instructions between the load and its dependent instructions. The unrelated instructions execute in parallel while the processor waits for the memory system to return the data. If the value is returned before the dependent operation reaches the execution stage of the pipeline, the operation completes in one cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may execute before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.

Interlocked Pipeline

In the execution of instructions, the Blackfin processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start execution before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for the memory-read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruction that uses the same register, all of them execute, and the overall throughput of the processor is improved.

Ordering of Loads and Stores

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations—even the order in which these events occur—may not align with how they appear in the sequence of the program source code. All that is guaranteed is:

- Load operations will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a

read that occurs in the program source code after a write in the program flow to actually return its value before the write is completed. This ordering provides significant performance advantages in the operation of most memory instructions. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

When writing to or reading from non memory locations such as I/O device registers, the order of how read and write operations complete is often significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the write buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these effects could cause undesirable side effects in the intended operation of the program and peripheral. To ensure that these effects do not occur in code that requires precise (strong) ordering of load and store operations, synchronization instructions (`CSYNC` or `SSYNC`) should be used.

Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential writes to an I/O device for setup and control, use the core or system synchronization instructions, `CSYNC` or `SSYNC`, respectively.

The `CSYNC` instruction ensures all pending core operations have completed and the core buffer (between the processor core and the L1 memories) is flushed before proceeding to the next instruction. Pending core operations may include any pending interrupts, speculative states (such as branch predictions), or exceptions.

Consider the following example code sequence:

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

In the example code, the CSYNC instruction ensures:

- The conditional branch (IF CC JUMP away_from_here) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.
- All pending interrupts or exceptions have been processed before CSYNC completes.
- The load is not fetched from memory speculatively.

The SSYNC instruction ensures that all side effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of CSYNC, the SSYNC instruction flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement before SSYNC completes.

Speculative Load Execution

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory-read operation for a subsequent load instruction usually has no undesirable side effect. In some code sequences, such as a conditional branch instruction followed by a load, performance may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved. For example,

```
            IF CC JUMP away_from_here
            R0 = [P2];
            …
away_from_here:
```

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory returns the correct value earlier than if the operation were stalled until the branch condition was resolved.

However, in the case of an I/O device, this could cause an undesirable side effect for a peripheral that returns sequential data from a FIFO or from a register that changes value based on the number of reads that are requested. To avoid this effect, use synchronizing instructions (`CSYNC` or `SSYNC`) to guarantee the correct behavior between read operations.

Store operations never access memory speculatively, because this could cause modification of a memory value before it is determined whether the instruction should have executed.

Conditional Load Behavior

The synchronization instructions force all speculative states to be resolved before a load instruction initiates a memory reference. However, the load instruction itself may generate more than one memory-read operation, because it is interruptible. If an interrupt of sufficient priority occurs between the completion of the synchronization instruction and the completion of the load instruction, the sequencer cancels the load instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory-read cycle was initiated before the load was canceled, and this would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side effects. However, for some memory-mapped devices, such as peripheral data FIFOs, reads are destructive. Each time the device is read, the FIFO advances, and the data cannot be recovered and re-read.

When accessing memory-mapped devices that have state dependencies on the number of read or write operations on a given address location, disable interrupts before performing the load or store operation.

Working With Memory

This section contains information about alignment of data in memory and memory operations that support semaphores between tasks. It also contains a brief discussion of MMR registers and a core MMR programming example.

Alignment

Nonaligned memory operations are not directly supported. A nonaligned memory reference generates a misaligned access exception event (see "System Interrupts" on page 6-1). However, because some data streams (such as 8-bit video data) can properly be nonaligned in memory, alignment exceptions may be disabled by using the `DISALGNEXCPT` instruction. Moreover, some instructions in the quad 8-bit group automatically disable alignment exceptions.

Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the `FLUSH` instruction (see "Data Cache Control Instructions" on page 3-41), and/or explicit line invalidation through the core MMRs (see "Data Test Registers" on page 3-42).

Atomic Operations

The processor provides a single atomic operation: TESTSET. Atomic operations are used to provide non interruptible memory operations in support of semaphores between tasks. The TESTSET instruction loads an indirectly addressed memory half word, tests whether the low byte is zero, and then sets the most significant bit (MSB) of the low memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally nonzero, the instruction clears the CC bit. The sequence of this memory transaction is atomic—hardware bus locking ensures that no other memory operation can occur between the test and set portions of this instruction. The TESTSET instruction can be interrupted by the core. If this happens, the TESTSET instruction is executed again upon return from the interrupt.

The TESTSET instruction can address the entire 4 Gbyte memory space, but should not target on-core memory (L1 or MMR space) since atomic access to this memory is not supported.

The memory architecture always treats atomic operations as cache inhibited accesses even if the CPLB descriptor for the address indicates cache enabled access. However, executing TESTSET operations on cacheable regions of memory is not recommended since the architecture cannot guarantee a cacheable location of memory is coherent when the TESTSET instruction is executed.

Memory-Mapped Registers

The MMR reserved space is located at the top of the memory space (0xFFC0 0000). This region is defined as non-cacheable and is divided between the system MMRs (0xFFC0 0000–0xFFE0 0000) and core MMRs (0xFFE0 0000–0xFFFF FFFF).

If strong ordering is required, place a synchronization instruction after stores to MMRs. For more information, see "Load/Store Operation" on page 3-70.

All MMRs are accessible only in supervisor mode. Access to MMRs in user mode generates a protection violation exception. Attempts to access MMR space using DAG1 will generate a protection violation exception.

All core MMRs are read and written using 32-bit aligned accesses. However, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved. System MMRs may be 16 bits.

Accesses to nonexistent MMRs generate an illegal access exception. The system ignores writes to read-only MMRs.

Appendix A provides a summary of all core MMRs. Appendix B provides a summary of all system MMRs.

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Nonaligned access to MMRs generates an exception event. Listing 3-1 shows the instructions required to manipulate a generic core MMR.

Listing 3-1. Core MMR Programming

```
CLI R0;   /* stop interrupts and save IMASK */
P0 = MMR_BASE;   /* 32-bit instruction to load base of MMRs */
R1 = [P0 + TIMER_CONTROL_REG];   /* get value of control reg */
BITSET R1, #N;   /* set bit N */
[P0 + TIMER_CONTROL_REG] = R1;   /* restore control reg */
CSYNC;   /* assures that the control reg is written */
STI R0;   /* enable interrupts */
```

The `CLI` instruction saves the contents of the `IMASK` register and disables interrupts by clearing `IMASK`. The `STI` instruction restores the contents of the `IMASK` register, thus enabling interrupts. The instructions between `CLI` and `STI` are not interruptable.

Terminology

The following terminology is used to describe memory.

cache block The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit A memory access that is satisfied by a valid, present entry in the cache.

cache line Same as cache block. In this chapter, cache line is used for cache block.

cache miss A memory access that does not match any valid entry in the cache.

direct-mapped Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-way associative.

dirty or **modified** A state bit, stored along with the tag, indicating whether the data in the data cache line is changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean The state of a data cache line indicating the line is valid and the data contained in the line matches that in source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative Cache architecture in which each line can be placed anywhere in the cache.

index Address portion that is used to select an array element (for example, a line index).

invalid Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm Replacement algorithm, used by cache, that first replaces lines that have been unused for the longest time.

level 1 (L1) memory Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy The function used by the processor to determine which line to replace on a cache miss. Often, an LRU algorithm is employed.

set A group of *N*-line storage locations in the ways of an *N*-way cache, selected by the `INDEX` field of the address (see Figure 3-4 on page 3-15).

set associative Cache architecture that limits line placement to a number of sets (or ways).

tag Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

valid A state bit, stored with the tag, indicating the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Way An array of line storage elements in an *N*-way cache (see Figure 3-4 on page 3-15).

write-back A cache write policy, also known as copyback. The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced. Cache lines are allocated on both reads and writes.

write-through A cache write policy, also known as store through. The write data is written to both the cache line and to the source memory. The modified cache line is *not* written to the source memory when it is replaced. Cache lines must be allocated on reads, and may be allocated on writes (depending on mode).

4 ONE-TIME PROGRAMMABLE MEMORY

This chapter describes one-time-programmable (OTP) memory features of the ADSP-BF54x processor Blackfin processor.

This chapter includes the following sections:

- "OTP Memory Overview" on page 4-1
- "Error Correction" on page 4-6
- "OTP Access" on page 4-8
- "OTP Timing Parameters" on page 4-10
- "Callable ROM Functions for OTP ACCESS" on page 4-14
- "Programming and Reading OTP" on page 4-16
- "Write Protecting OTP Memory" on page 4-24
- "Accessing Private OTP Memory" on page 4-26
- "OTP Programming Examples" on page 4-26

OTP Memory Overview

The ADSP-BF54x processor processors include an on-chip, one-time-programmable memory array which provides 64K bits of non-volatile memory. This includes the array and logic to support read access and programming. A mechanism for error correction is also provided. Additionally, pages can be write protected.

OTP memory can be programmed through various methods, including software running on the Blackfin processor. The ADSP-BF54x processor processors provide C and assembly callable functions in the on-chip ROM to help the developer access the OTP memory.

The one -time-programmable memory is divided into two main regions. A 32K bit “public” unsecured region, which has no access restrictions; and a 32K bit “private” secured region with access restricted to authenticated code when operating in Secure Mode. For information about these modes, see “Secure State Machine” on page 16-7.

OTP allows developers to store both public and private data on-chip. A 64K by 1 bit array is available as shown in Figure 4-1. In addition to storing public and private data, it allows developers to store completely user-definable data, such as customer ID, product ID, and MAC address.

The public portion of OTP memory contains many “factory set only” values. Users are urged to exercise caution when writing to OTP memory and to consult the OTP memory map for details of Customer Programmable Settings (CPS) and factory reserved areas of this memory. See also “Factory Page Settings (FPS)” on page 17-14 and “Preboot Page Settings (PBS)” on page 17-14.

OTP Memory Map

The OTP is not part of the Blackfin linear memory map. It has a separate memory map as shown in Figure 4-1 on page 4-3 and Figure 4-2 on page 4-4. OTP memory is not accessed directly using the Blackfin memory map; rather, it is accessed through four 32-bit wide registers (`OTP_DATA3-0`) which act as the OTP memory read/write buffer.

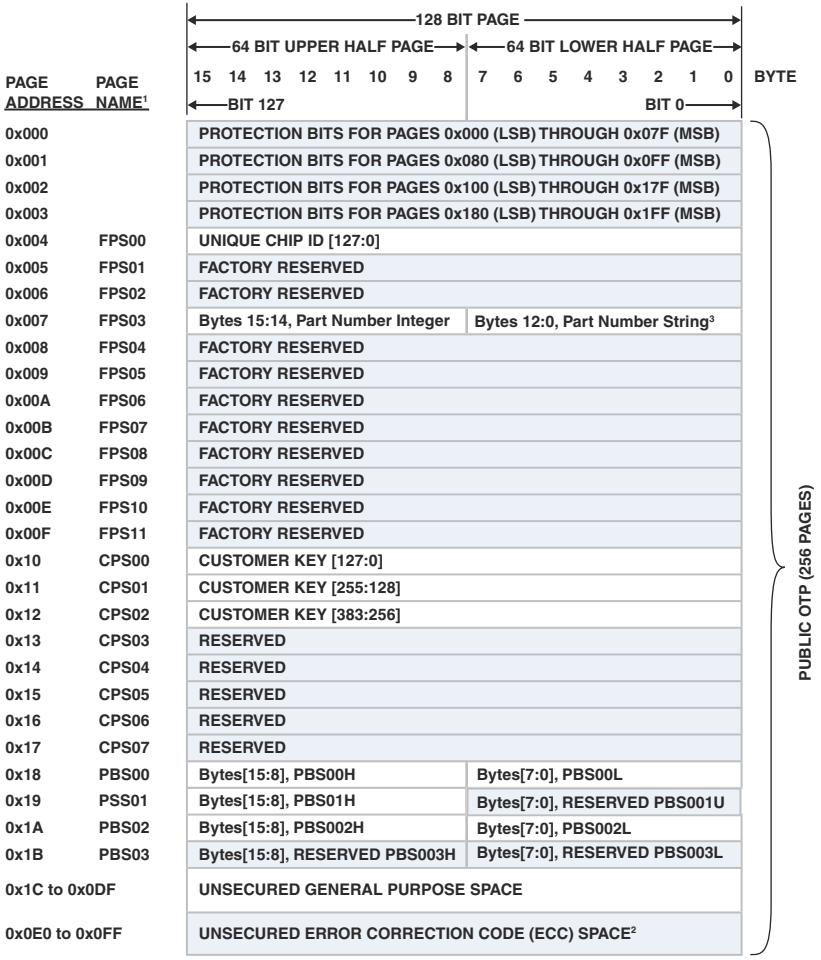
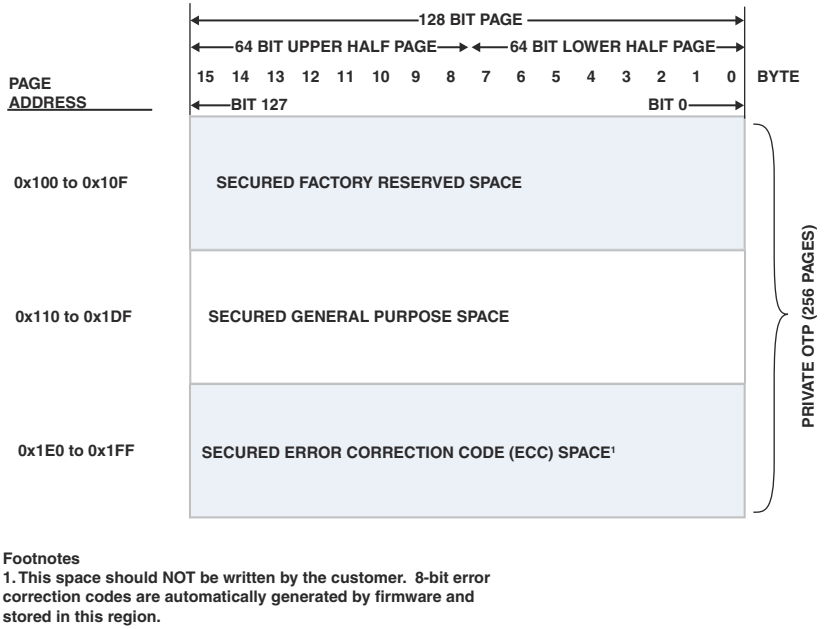


Figure 4-1. One-Time-Programmable (OTP) Public Memory Map



Footnotes
1. This space should NOT be written by the customer. 8-bit error correction codes are automatically generated by firmware and stored in this region.

Figure 4-2. One-Time-Programmable (OTP) Private Memory Map

For an OTP memory read, the `OTP_DATAx` registers contain the 16-byte result of the OTP memory access. For an OTP memory write, the `OTP_DATAx` registers contain 16 bytes of data to be written to the OTP memory.

The `OTP_DATA3-0` registers are organized into a 128 bit page as shown in Figure 4-3.

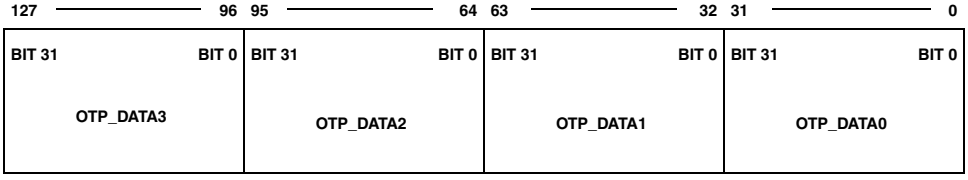


Figure 4-3. OTP_DATAx Registers

OTP memory ranges marked as “factory reserved” and “error correction code space” (see Figure 4-1 on page 4-3 and Figure 4-2 on page 4-4) must not be programmed by the developer. Customer programmable settings may be programmed by the developer.

Page-protection bits provide protection for each 128-bit page within the OTP. By default the OTP array bits are not set, and will read back as zero values if left unprogrammed. Programmed data values consist of zeroes and ones; therefore, after programming OTP memory, some bits will intentionally remain as zero values. The write-protect bits provide protection for the zero value bits to remain as zeroes and prevent future programming (inadvertent or malicious) from changing bit values from zero to one.

Pages 0x10, 0x11 and 0x12 hold the customer public key, which is used for Lockbox digital signature authentication. Refer to Chapter 16, “Security” for more information on Lockbox and how the public key is used.

OTP memory is logically arranged in a sequential set of 128-bit pages. Each OTP memory address refers to a 128-bit page. The ADSP-BF54x processor thus provides 512 pages of OTP memory.

To read or program the OTP memory, a set of functions are provided in the on-chip ROM. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

Error Correction

Error correction, provided in the on-chip ROM, can be used to ensure data integrity.

Error correction, when programmed into the OTP, calculates an 8-bit error correction code (ECC) for each 64-bit data word (half page). When this word is later read from OTP, its corresponding ECC is also read, and a data integrity check is performed. If the check fails, the ECC can be used to attempt error correction on the data word. The error correction algorithm depends on the type of error, as shown in Table 4-1.

Table 4-1. Hamming Code Single Error Corrections, Double Error Detection

No. of bad bits in data word	Error(s) Detected?	Error(s) Corrected?
0	N/A	N/A
1	Yes	Yes
2	Yes	No
3 or more	No	No

Error Correction Policy

1. Error correction requires that the OTP space be written and read in 64-bit widths. Firmware will only support writing or reading half of an OTP page.
2. Error correction is used to correct data in all pages of OTP space except the protection pages (0x0 to 0x3) and the ECC pages themselves. See "OTP Access" on page 4-8 for more information.
3. Firmware will generate and program the 8-bit ECC fields as mapped in Table 4-2 and Table 4-3.

4. The developer is responsible for locking both the data page(s) *and* the ECC page(s) after all programming is complete.
5. Pages 0x04 to 0x0F are reserved for factory use. Therefore, pages 0x004 to 0x00F, 0x0E0, and 0x0E1 are locked when devices leave the Analog Devices Inc. factory.

Table 4-2. Mapping for Storage of Error Correction Codes for Unsecured OTP Space

Page	**Byte**							
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
0x0E0	0x007U	0x007L	0x006U	0x006L	0x005U	0x005L	0x004U	0x004L
0x0E1	0x00FU	0x00FL	0x00EU	0x00EL	0x00DU	0x00DL	0x00CU	0x00CL
0x0E2	0x017U	0x017L	0x016U	0x016L	0x015U	0x015L	0x014U	0x014L
....								
0x0FB	0x0DFU	0x0DFL	0x0DEU	0x0DEL	0x0DDU	0x0DDL	0x0DCU	0x0DCL
Page	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0x0E0	Unused	Unused	Unused	Unused	Unused	Unused	Unused	Unused
0x0E1	0x00BU	0x00BL	0x00AU	0x00AL	0x009U	0x009L	0x008U	0x008L
0x0E2	0x013U	0x013L	0x012U	0x012L	0x011U	0x011L	0x010U	0x010L
....								
0x0FB	0x0DBU	0x0DBL	0x0DAU	0x0DAL	0x0D9U	0x0D9L	0x0D8U	0x0D8L

Table 4-3. Mapping for Storage of Error Correction Codes for Secured OTP Space

Page	Byte							
	15	14	13	12	11	10	9	8
0x1E0	0x107U	0x107L	0x106U	0x106L	0x105U	0x105L	0x104U	0x104L
0x1E1	0x10FU	0x10FL	0x10EU	0x10EL	0x10DU	0x10DL	0x10CU	0x10CL
0x1E2	0x117U	0x117L	0x116U	0x116L	0x115U	0x115L	0x114U	0x114L
....								
0x1FB	0x1DFU	0x1DFL	0x1DEU	0x1DEL	0x1DDU	0x1DDL	0x1DCU	0x1DCL
Page	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0x1E0	0x103U	0x103L	0x102U	0x102L	0x101U	0x101L	0x100U	0x100L
0x1E1	0x10BU	0x10BL	0x10AU	0x10AL	0x109U	0x109L	0x108U	0x108L
0x1E2	0x113U	0x113L	0x112U	0x112L	0x111U	0x111L	0x110U	0x110L
....								
0x1FB	0x1DBU	0x1DBL	0x1DAU	0x1DAL	0x1D9U	0x1D9L	0x1D8U	0x1D8L

OTP Access

The ADSP-BF54x processor on-chip ROM contains the functions for initializing OTP timing parameters, reading the OTP memory, and programming the OTP memory. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

These functions are callable from C or assembly application code. Use only these functions for accessing OTP memory. Directly accessing memory locations within OTP memory by other means is not supported.

The existing ECC in ROM is known as “Hamming [72,64]”. This is a 64-bit data and an 8-bit ECC field—for a 1-bit correction and 2-bit error detection scheme.

The ROM-based OTP read/write API *must* be used for all OTP data accesses (see limited exceptions below). The ROM code incorporates the *only* ECC method supported by Analog Devices Inc. Direct access of OTP data without using error correction is not supported.

Exceptions: The only bits that do not use ECC are page lock bits (first four pages) and the preboot invalidate bits. See “Preboot Page Settings (PBS)” on page 17-14.

Analog Devices Inc. does not support any ECC other than the ECC provided by Analog Devices Inc. in the ROM API. All attempts to implement other schemes are not guaranteed or supported by Analog Devices Inc.

OTP memory programming is done serially under software control. Since the unprogrammed OTP memory value defaults to zero, only those bits whose value is intended to be “1” have to be programmed. Write-protect bits (see Figure 4-1 on page 4-3) can be set for each 128-bit page within OTP memory to protect areas of OTP memory that have been programmed, or areas left unprogrammed that developers wish to remain unchanged. Each write-protect bit on a per page basis, when set, will prevent further programming attempts to OTP memory.

The ADSP-BF54x processor Blackfin processor can program OTP through software code executing directly on the Blackfin processor. A charge pump residing on-chip is used to apply the voltage levels appropriate for programming OTP memory. OTP programming code can be loaded into the processor during JTAG emulation, through the DMA, and through all supported boot methods.

OTP memory can only be written once (changing a bit from 0 to 1). Once a bit has been changed from a 0 to a 1, it cannot be changed back to 0. The write-protect bits prevent OTP memory that has already been programmed from having any bits that are meant to remain as 0 value later programmed to a value of 1.

To ensure reliable OTP programming, before accessing OTP memory, see *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for specifications on `VDDINT` and `VDDEXT` voltage levels. OTP timing parameters must be set before attempting any write accesses to OTP.

OTP Timing Parameters

To read and program the OTP memory reliably, the OTP timing parameters must be set correctly before accessing OTP memory. All of the timing parameters are bit fields in the `OTP_TIMING` register, as shown in Figure 4-4 on page 4-13. The function, `bfrom_OtpCommand()`, provided in the on-chip ROM, is used to program the timing parameters.

OTP timing parameters must be set by using the `bfrom_OtpCommand()` as described in "bfrom_OtpCommand" on page 4-14. OTP read accesses may use the OTP timing default reset value (`OTP_TIMING` = 0x0000 1485 for reset). Using the OTP timing default reset value for writes results in write errors, since this timing value is not appropriate for write accesses.

Insufficient voltage/current provided to OTP during write access or incorrect OTP timing parameters may return an 0x11 error code (multiple bad bits in 64 bit data) during OTP writes. Subsequent reads from this page return 0.

The OTP timing parameters consist of several fields which are combined together to form one value which is then passed as an argument to the `bfrom_OtpCommand()` function. The developer must calculate a value for two fields based upon the `SCLK` frequency at which the OTP access will be

performed. These calculated values are then combined with a third field whose value is provided by Analog Devices Inc. to arrive at the setting appropriate for the access.

The OTP timing parameters are comprised of three values as follows:

`OTP_TIMING[7:0]` = `OTP_TP1` = 1000/sclk_period

`OTP_TIMING[14:8]` = `OTP_TP2` = 400/(2*sclk_period)

`OTP_TIMING[31:15]` = `OTP_TP3` = 0x0A008

The `OTP_TP3` field is specified by Analog Devices Inc. and must be used to ensure reliable OTP write accesses. The user calculated fields must be combined with the `OTP_TP3` value as shown in the following examples.

Example calculations are shown in the following sections based upon voltages specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*. The calculations depend upon user-defined `SCLK` frequency of operation. (Refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for actual specifications. Do not rely on the specifications quoted in the examples.)

OTP Timing Calculations for SCLK = 100 MHz

For `SCLK` = 10ns (100 MHz), the following field calculations are needed to determine the OTP timing argument for the `bfrom_OtpCommand()` call.

`OTP_TP1` = 1000/`SCLK` = 1000/10 = 0x64	0x0000 0064
`OTP_TP2` = 400/(2*`SCLK`) = 400/(2*10) = 0x14	0x0000 1400
`OTP_TP3` = (constant)	0x0A00 8xxx
Calculated OTP timing parameter value	0x0A00 9464

Example code for the API call (in C) is:

```
/* Initialize OTP access settings */
/* Proper access settings for VDDINT = 1V, SCLK = 100 MHz */
const u32 OTP_init_value = 0x0A009464;
return_code = bfrom_OtpCommand  (OTP_INIT, OTP_init_value);
```

OTP Timing Calculations for SCLK = 50 MHz

For SCLK = 20.0ns (50 MHz), the following field calculations are needed to determine the OTP timing argument for the bfrom_OtpCommand() call.

OTP_TP1 = 1000/SCLK = 1000/20.0 = 0x32	0x0000 0032
OTP_TP2 = 400/(2*SCLK) = 400/(2 * 20.0) = 0xA	0x0000 0A00
OTP_TP3 = (constant)	0x0A00 8xxx
Calculated OTP timing parameter value	0x0A00 8A32

Example code for the API call (in C) is:

```
/* Initialize OTP access settings */
/* Proper access settings for VDDINT = 1V, SCLK = 50 MHz */
const u32 OTP_init value = 0x0A008A32;
return_code = bfrom_OtpCommand(OTP_INIT, OTP_init_value);
```

OTP Timing Calculations for SCLK = 40 MHz

For SCLK = 25.0ns (40 MHz), the following field calculations are needed to determine the OTP timing argument for the bfrom_OtpCommand() call.

OTP_TP1 = 1000/SCLK = 1000/25.0 = 0x28	0x0000 0028
OTP_TP2 = 400/(2*SCLK) = 400/(2*25.0) = 0x8	0x0000 0800
OTP_TP3 = (constant)	0x0A00 8xxx
Calculated OTP timing parameter value	0x0A00 8828

Example code for the API call (in C) is:

```
/* Initialize OTP access settings  */
/* Proper access settings for VDDINT = 1V, SCLK = 40 MHz */
const u32 OTP_init_value = 0x0A008828
return_code = bfrom_OtpCommand(OTP_INIT, OTP_init_value);
```

OTP_TIMING Register

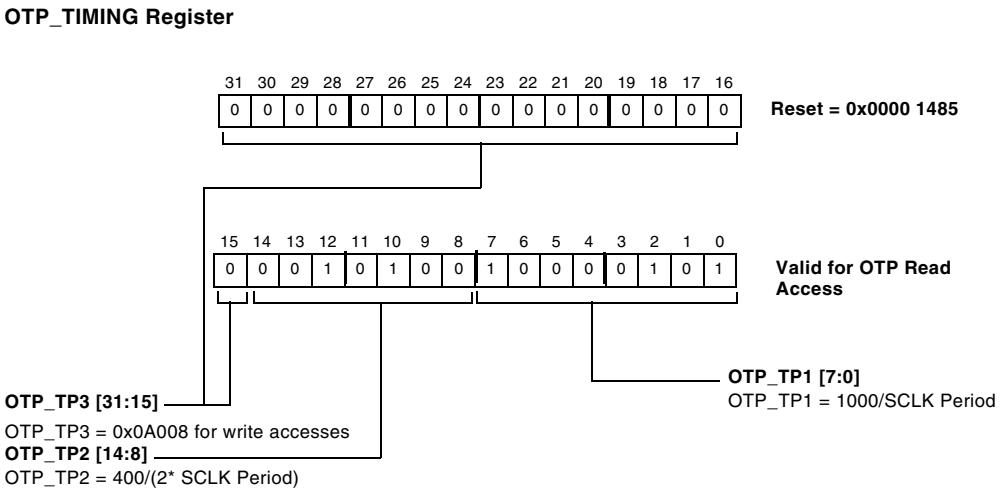


Figure 4-4. OTP_TIMING Register

Callable ROM Functions for OTP ACCESS

The following functions are provided in the ADSP-BF54x processor processor on-chip ROM to support OTP access.

Initializing OTP

This section describes the `bfrom_OtpCommand()` function for OTP memory controller setup. The prototype and macros that decode the function's returns are supplied by the `bfrom.h` header file which is located in the CCES or VisualDSP++ installation directory. The meaning of the error code is described in section "Error Codes" on page 4-21.

bfrom_OtpCommand

This function sets up the OTP controller.

Entry address: 0xEF00 0018

Arguments:

`R0`: command (`dCommand`)
`OTP_INIT`
`OTP_CLOSE`

`R1`: timing value to be programmed (`dValue`),
not used for `OTP_CLOSE`

C Prototype:

```
u32 bfrom_OtpCommand(u32 dCommand, u32 dValue);
```

Return code:
`bfrom_OtpCommand()`
currently always returns with "0".

The first input parameter is a mnemonic label specifying the command. The second parameter is a generic value that is passed as the argument for the requested command. The second parameter is optional and may be an integer value or (through opportune casting) a pointer or a pointer to an extension structure.

There are two commands:

- `OTP_INIT`
  Sets the required timing value (register `OTP_TIMING`) to "value"
- `OTP_CLOSE`
  Reinitializes the OTP controller. If desired, this can be called by the user before exiting Secure Mode. The value parameter may be specified as "0" or "NULL" with `OTP_CLOSE`.

In the example above ("OTP Timing Calculations for SCLK = 100 MHz" on page 4-11), the OTP timing parameter was calculated to be 0x0A00 9464. Listing 4-1 on page 4-15 shows a sample of C code that uses the `bfrom_OtpCommand()` function to program this parameter.

Listing 4-1. Programming the `bfrom_OtpCommand()` Function

```
#include <bfrom.h>
#define OTP_TIMING_PARAM (0x0A009464)

u32 Otp_Timing_Param_Init()
   {
   u32 otp_timing_parameter;
   u32 = RetVal;
   otp_timing_parameter = OTP_TIMING_PARAM;
   RetVal = bfrom_OtpCommand(OTP_INIT, otp_timing_parameter);
/* (equivalently, with a variable):  */
   RetVal = bfrom_OtpCommand(OTP_INIT, OTP_TIMING_PARAM);
   return RetVal;
   }
```

Listing 4-2 on page 4-16 shows another example.

Listing 4-2. Programming the `bfrom_OtpCommand()` Function

```
/* timing parameter  */
const u32 init_value = 0x0A009464;
/* call sets OTP_TIMING register  */
RetVal = bfrom_OtpCommand(OTP_INIT, init_value);

/* call sets OTP_TIMING register  */
RetVal = bfrom_OtpCommand(OTP_INIT, 0x0A009464);

/* call clears OTP controller and data registers  */
RetVal = bfrom_OtpCommand(OTP_CLOSE, NULL);
```

The prototype of `bfrom_OtpCommand()` is included in the `bfrom.h` header file installed with the VisualDSP++ 5.0 or CrossCore Embedded Studio IDE. The macro `OTP_INIT` is defined in `bfrom.h` as well.

Programming and Reading OTP

This section describes the `bfrom_OtpRead()` and `bfrom_OtpWrite()` read and write functions. The prototypes and macros that decode the function's returns are supplied by the `bfrom.h` header file which is located in the CCES or VisualDSP++ installation directory. The meaning of the error code is described in section "Error Codes" on page 4-21.

bfrom_OtpRead

This function is used to read 64-bit OTP half-pages using error correction.

Entry address: 0xEF00 001A

Arguments:

`R0`: OTP page address (`dPage`)

`R1`: Flags (`dFlags`)
```
OTP_LOWER_HALF
OTP_UPPER_HALF
OTP_NO_ECC
```

`R2`: Pointer (`*pPageContent`) to 64-bit memory struct (long long) where the data that is read will be placed

C prototype:
```
u32 bfrom_OtpRead (u32 dPage, u32 dFlags, u64 *pPageContent);
```

Return code:
`R0`: error or warning code (see Table 4-4)

This function reads a half-page and stores the content in the 64-bit variable pointed to by the page parameter `R2`. The `*pPageContent` pointer defines the address. The flags parameter R1 defines whether the upper or the lower half page is to be read.

The default reset value for `OTP_TIMING` (0x0000 1485) may be used for all read accesses without requiring a new value be programmed before performing read accesses. Programming a value valid for write accesses will also allow read accesses.

The use of flag parameter `OTP_NO_ECC` is not recommended for any OTP read access because it bypasses error correction code support. It is available only for diagnostic purposes.

bfrom_OtpWrite

This function writes to (programs) a half-page with the content in the 64-bit variable pointed to by the `R2` parameter.

Entry address: 0xEF00 001C

Arguments:

`R0`: OTP page address (`dPage`)

`R1`: Flags (`dFlags`)
```
OTP_LOWER_HALF
OTP_UPPER_HALF
OTP_NO_ECC
OTP_LOCK
OTP_CHECK_FOR_PREV_WRITE
```

`R2`: Pointer (`*pPageContent`) to 64-bit memory struct (long long) that contains the data to be written to OTP memory

C Prototype:
```
u32 bfrom_OtpWrite (u32 dPage, u32 dFlags, u64 *pPageContent);
```

Return code:
`R0`: error or warning code, see Table 4-4.

The `dFlags` parameter defines whether the upper or the lower half page is to be written to and if the target half page should be checked for a previously written value before a write attempted. Additionally, a page can be locked (permanently protected against further writes).

When performing pure lock operations, the half-page parameter is not required and it makes no difference which half-page is specified if this parameter is included in the function call.

To reduce the probability of inadvertent writes to OTP pages, this function checks for a valid OTP write timing setting in the OTP_TIMING register. Specifically, bits [31:15] must not be equal to zero. Calls to the write routine when this field is equal to zero cause an access violation error and the requested action is not performed. The developer can use this mechanism to protect against inadvertent writes by calling the bfrom_OtpCommand (OTP_INIT, …) function with appropriate values for reads only and for read/write accesses. The developer is also free to ignore this mechanism by calling bfrom_OtpCommand (OTP_INIT, …) only once for read/write access.

When the flag OTP_CHECK_FOR_PREV_WRITE is *not* specified, a previously written value will be overwritten, both in the ECC and data fields for any unlocked page where a write access is performed. Once a bit was set to "1" it cannot be reset to "0" by the new write operation. This means that if the new value is different from the previous one, there will be multiple bit errors, in either or both the ECC and data fields.

Since the ECC field is written first by the ROM function, a multiple bit error will abort the operation without writing the new data value to the OTP data page.

Note also that multiple bit errors have a statistical chance of not being detected as such. Therefore this mode of operation should not be used, or used with caution.

The flag OTP_CHECK_FOR_PREV_WRITE should always be used when performing write accesses to OTP with the bfrom_OtpWrite() function.

If the flag OTP_CHECK_FOR_PREV_WRITE is specified in the call, a write to a previously programmed page causes dedicated error messages and will not be performed.

Specifically, errors are generated as follows.

- The 64-bit data and the 8-bit ECC field are read and the total number of “1”s is counted.
- If this number is equal to or greater than 2, the error flag `OTP_PREV_WR_ERROR` is returned and the write operation is not performed.
- If the number is 0, the page is certainly blank and the write is performed.
- If the number is 1, a more thorough check is performed.

  If the “1” is in the ECC field, an error flag `OTP_SB_DEFECT_ERROR` is returned and the write is not performed.

  If the “1” is in the data field, it is determined whether the value to be written contains a “1” in the same position.

  If so, the write is performed.

  If not, the error flag `OTP_SB_DEFECT_ERROR` is returned and the write is not performed. This error code warns the user that it could be a single-bit defect in the page. The user can then decide whether to use this page regardless (by repeating the call without the `OTP_CHECK_FOR_PREV_WRITE` flag) or skip this page.

The `OTP_CHECK_FOR_PREV_WRITE` flag is ignored when a pure lock operation is requested (for example, a `OTP_LOCK` flag is set and `*pPageContent` = null). It would then be unnecessary and harmless to specify this flag.

The `OTP_CHECK_FOR_PREV_WRITE` flag is not ignored when doing a lock operation after a write (for example, `OTP_LOCK` plus write in the same call and `*pPageContent` = null).

If the flag parameter for the write operation is OR'ed with the `OTP_LOCK` flag—the write operation, if successful, will be immediately followed by setting the protection bit for the requested full 128-bit page.

A special case for `OTP_LOCK` is the following. If the third parameter is null, this call will lock a page without writing any data value to it (pure lock function). Note that in this case, "page" can span all pages from 0x000 to 0x1FF. *This is the only way to lock the ECC pages themselves.*

The use of flag parameter `OTP_NO_ECC` is only supported in write operations for write-protection/page-locking, or to set the preboot invalidate bits (see "Preboot" on page 17-11). The preferred method for locking pages is to use the `OTP_LOCK` parameter in the `bfrom_Otp_Write` function (see "Write Protecting OTP Memory" on page 4-24). Bypassing error correction for OTP writes may cause loss of OTP data integrity and is not supported.

ECC must be used for all OTP accesses other than the limited exceptions described previously.

Error Codes

This section describes the error codes that may be returned by the API functions. These are shown in Figure 4-5 and listed in Table 4-4.

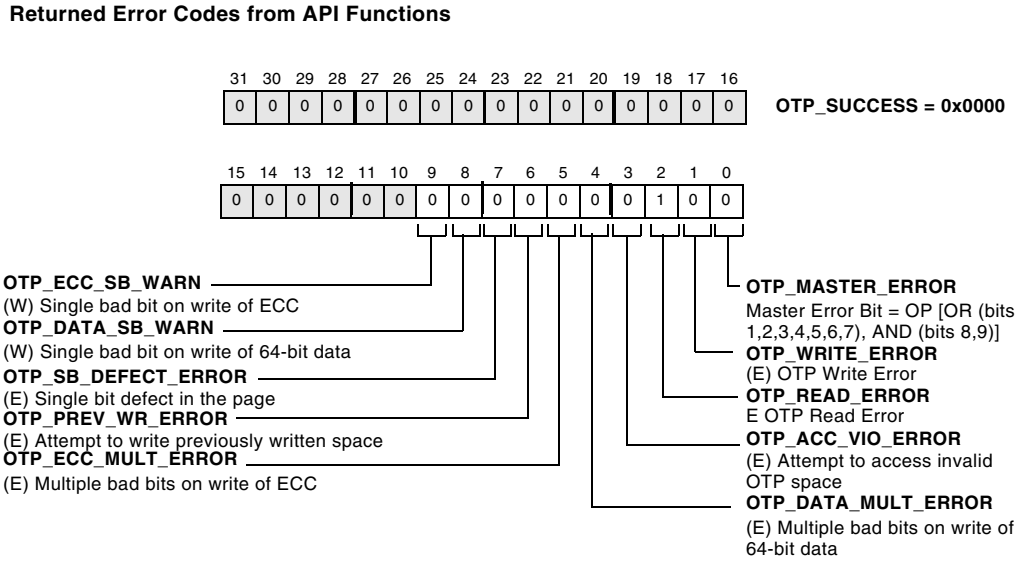


Figure 4-5. Returned Error Codes from API Functions

Table 4-4. Returned Error Codes from API Functions

Bit	Name	Example Return Value	Definition
N/A	OTP_SUCCESS	0x0	No error
0	OTP_MASTER_ERROR	0x1	Master error bit = OR [OR (bits 1,2,3,4,5,6,7), AND (bits 8,9)]
1	OTP_WRITE_ERROR	0x3	OTP write error
2	OTP_READ_ERROR	0x5	OTP read error
3	OTP_ACC_VIO_ERROR	0x9	Error on attempt to access invalid OTP space
4	OTP_DATA_MULT_ERROR	0x11	Error for multiple bad bits when writing 64 bit data
5	OTP_ECC_MULT_ERROR	0x21	Error for multiple bad bits when writing ECC
6	OTP_PREV_WR_ERROR	0x41	Error on attempt to write previously written space
7	OTP_SB_DEFECT_ERROR	0x81	Error for single bit defect in the page
8	OTP_DATA_SB_WARN	0x100	Warning about single bad bit when writing 64 bit data
9	OTP_ECC_SB_WARN	0x200	Warning about single bad bit when writing ECC

`bfrom_OtpCommand()` always returns with “0”.

`bfrom_OtpRead()` returns with an error when any of the bits6–2 are set or both bits[9:8] are set. The `OTP_MASTER_ERROR` bit is also set. It returns with a warning, if only one of the bits [9:8] is set.

`bfrom_OtpWrite()` returns with an error when any of the bits7–1 are set or both bits[9:8] are set. The `OTP_MASTER_ERROR` bit is also set. It returns with a warning, if only one of the bits [9:8] is set.

Write Protecting OTP Memory

As shown in Figure 4-1, a small portion of OTP memory is reserved for write-protect bits (“write-protect” is synonymous with “page-protect” in this context). After programming OTP memory, the programmer can use these protection bits to “lock” the page that was just programmed by setting the write-protect bit corresponding to the OTP data page. Once the write-protect bit is set and the lock is in place, further attempts to write to that page are not allowed, which results in an error. Page protect bits can also be set to prevent programming of unwritten OTP pages. Once an OTP page is page-protected, the write protection cannot be reversed and no further write accesses can be made to the protected page(s).

There are four pages reserved for the write-protection bits. Pages 0x0 through 0x3 contain the 512 write-protect bits—one bit for each of the 512 data pages within OTP memory. The first two write-protect bit pages (pages 0x0 and 0x1) correspond to the public (non-secure) regions of the OTP map. The other two write-protect bit pages (0x2 and 0x3) correspond to protection of the private (secure) regions of the OTP map. The processor does not need to be operating in Secure Mode in order to program the protection pages associated with the secure OTP regions. All protection bits can be written in any security state including Open Mode.

While reads and writes access a half-page at a time, setting a protection bit for a page effectively locks an entire page from future write accesses (both lower and upper half page). The programmer must make sure that a full 128-bit OTP page is programmed, or that no future programming will be needed before setting the write-protect bit for that page.

To lock a page (P), set the write-protect bit (WPB) and the page (WPP) where it resides as follows.

```
WPP = P >> 7;
WPB = P & 0x7f;
```

However, manual calculation is generally not needed because the `bfrom_OtpWrite()` function can be used to lock pages (see "OTP Programming Examples" on page 4-26).

Listing 4-3. Lock a Page

```
/* lock page (note third parameter equals NULL) */

return_code = bfrom_OtpWrite(0x01C, OTP_LOCK, NULL);
```

Locking a single ECC (error correction code) page locks the correction codes for eight OTP data pages (16 half pages). Since a 64-bit half-page access must be performed to write protect the ECC page and every 8-bits within an ECC page is a parity correction code corresponding to a 64-bit half-page of data in OTP—a full 128-bit ECC page holds the correction codes for eight full 128-bit pages of data in OTP, or 16 half-pages. Pages can only be locked as full 128-bit pages even though read/write accesses may occur at 64-bit half-page granularity. Locking a single ECC page prevents further write access to the corresponding eight OTP data pages.

ECC (error correction code) space cannot be written-to directly.

For example, locking ECC page 0xFB will result in locking the error correction parity data associated with the 16 data pages in the range of 0x0D8 – 0x0DF.

Listing 4-4. Lock ECC Page Only

```
/* Only Lock ECC code page */

return_code = bfrom_OtpWrite(0xFB, OTP_LOCK, NULL);
```

No further write accesses to the ECC page 0xFB or corresponding data pages 0x0D8 – 0x0DF will be allowed following write protection of the ECC page in this example.

Bits [3:0] of OTP page 0 are the write-protect bits for the first four OTP pages, which contain the write-protect bits. Setting these bits prevents the other write-protect bits from being set, which disables the write protection mechanism of the remaining user-programmable OTP pages.

Accessing Private OTP Memory

To read or write to the private area of OTP memory, the processor must be operating in Secure Mode and the `OTPSEN` bit in the `SECURE_SYSSWT` register must be set to 1 to enable secured OTP access. For more information about Security, Secure Mode and the Secure State Machine, see "Secure State Machine" on page 16-7).

OTP Programming Examples

The following sequence is recommended for accessing OTP memory.

1. Initialize the OTP array by calling `bfrom_OtpCommand()`.
2. Perform a OTP read or write access by calling the `bfrom_OtpRead()` or `bfrom_OtpWrite()` function.
3. When OTP read/write access is complete —call the `bfrom_OtpCommand()` function with the `OTP_CLOSE` parameter to re-initialize the OTP controller.
4. Initialize the OTP array by calling `bfrom_OtpCommand()` again for the next OTP access.
5. Repeat steps 1–3 for subsequent OTP accesses.

Enable Access to Private OTP

To enable access to private OTP memory space while operating in Secure Mode, use the following code.

Listing 4-5. Enable Access to Private OTP

```
/* Enable private OTP access */

*pSECURE_SYSSWT = *pSECURE_SYSSWT | OTPSEN;
SSYNC();
...
```

Enable Access to Private OTP and Enable JTAG Emulation in Secure Mode

To enable access to private OTP memory space by using `OTPSEN` while operating in Secure Mode, use the following code.

Listing 4-6. Enable Access to Private OTP and Enable JTAG Emulation in Secure Mode

```
/* Enable JTAG and private OTP access */

*pSECURE_SYSSWT = *pSECURE_SYSSWT & (~EMUABL) | OTPSEN;
SSYNC(0);
...
```

Read Public OTP Memory and Print to Console

To read pages 0x4 through 0xDF in public OTP memory space and print results to VisualDSP++ console, use the following code.

Listing 4-7. Read Public OTP Memory and Print to Console

```
#include <blackfin.h>
#include <bfrom.h>
u32 return_code, i;
u64 value;

/*  Initialize OTP timing parameter */
/*  Proper timing for VDDINT = 1v, CCLK, SCLK = 100MHz */
const u32 OTP_init_value = 0x0A009464;
return_code = bfrom_OtpCommand(OTP_INIT, OTP_init_value);
...
for (i= 0x004; i,0x0xE0; i++)
    {
return_code = bfrom_OtpRead(i, OTP_LOWER_HALF, &value);

printf("page: 0x%03xL, Content ECC: 0x%01611x,
returncode:  0x%03x \n", i, value, return_code);

return_code = bfrom_OtpRead(i, OTP_UPPER_HALF, &value);

printf("page: 0x%03xH, Content ECC: 0x%01611x,
returncode:  0x%03x \n", i, value, return_code);
    }
```

OTP Write to Single Page Using Two Half Page Accesses

To write and lock a single OTP page and return the results to the IDE console using `printf`, use the following code.

Listing 4-8. Perform OTP write to a single page via two 64-bit (half-page) accesses

```
#include <blackfin.h>   #include <bfrom.h>
u64 value;
u32 return_code;

return_code = bfrom_OtpWrite(0x01C, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE, &testdata);

printf("WRITE page: 0x%03xL, Content ECC: 0x%01611x,
returncode: 0x%03x \n", 0x1C, testdata, return_code);

return_code = bfrom_OtpWrite(0x01C, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE | OTP_LOCK, &testdata);

printf("WRITE page: 0x%03xH, Content ECC: 0x%01611x,
returncode: 0x%03x \n", 0x1C, testdata, return_code);
```

Locking a page will lock the full 128-bit page, even though the examples above access OTP with 64-bit half-page granularity. This is the finest level of granularity that is allowed due to the OTP error correction. The page should be locked, only after both the lower and upper portion of the page have been written. Note that in Listing 4-8 the page lock operation is performed on the second and final access to the page.

Lock Page Without Writing Any Data

To lock specific OTP pages in a separate access, after data values have been separately written, using the following code. OTP pages are typically locked in order to protect them from being overwritten or to prevent inadvertent or malicious tampering.

Listing 4-9. Lock a Page Without Writing Any Data

```
#include <blackfin.h>
#include <bfrom.h>   u64 value;
u32 return_code;

//  Initialize OTP timing parameter
//  Proper timing for VDDINT = 1V, CCLK, SCLK = 100MHz
const u32 OTP_init_value = 0x0A009464;

return_code = bfrom_OtpCommand(OTP_INIT, OTP_init_value);

return_code = bfrom_OtpWrite(0x01C, OTP_LOCK, NULL);
```

5 EXTERNAL BUS INTERFACE UNIT

The external bus interface unit (EBIU) provides a glueless interface to a variety of external memories. The EBIU supports both synchronous and asynchronous memories. The synchronous interface supports dual data rate (DDR) SDRAM memories. The asynchronous interface supports memories such as SRAM and flash memories including synchronous NOR flash.

The synchronous interface is controlled by a DDR controller. The asynchronous interface is controlled by the asynchronous memory controller (ASYNC). The asynchronous interface is further shared by an on-chip NAND flash controller and an ATAPI controller. The ATAPI and the NAND flash controllers are not part of EBIU; they just share the asynchronous interface pins. An asynchronous pin control module (APCM) controls and arbitrates the asynchronous interface between the ASYNC, NAND, and ATAPI controllers.

The chapter includes the following sections:

- "General Overview" on page 5-2
- "DDR Arbitration" on page 5-11
- "DDR SDRAM Controller" on page 5-15
- "DDR SDRAM Memory Interface" on page 5-18
- "DDR Registers" on page 5-23
- "DDR Metrics Control Registers" on page 5-44

- "Asynchronous Memory Interface" on page 5-53
- "Asynchronous Memory Interface Control Registers" on page 5-57

General Overview

The EBIU services requests for external memory from the Blackfin core and from three on-chip DMA controllers (DMAC0, DMAC1, and USB DMA). An address decoder inside EBIU determines whether the request is serviced by the DDR memory controller or the asynchronous memory controller and routes the requests to the appropriate controller. Requests from different sources are prioritized based on a programmable priority scheme.

The EBIU is clocked by the system clock (`SCLK`), which runs at a maximum frequency that is specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*. All DDR SDRAM memories interfaced to the device operate at `SCLK` frequency.

The external memory space is shown in Figure 5-1. Two of the memory regions are dedicated to DDR SDRAM. The DDR SDRAM interface timing and the size of each DDR SDRAM region are programmable. Each external DDR SDRAM bank can be populated up to 256M bytes. The start address of bank 0 is 0x0000 0000 and the start address of bank 1 follows contiguously from the previous bank. Depending upon the memory configuration, the area from the end of bank 1 to address 0x2000 0000 is reserved.

The next four regions are dedicated to support asynchronous memories. Each asynchronous memory region can be independently programmed to support different memory device characteristics. Each region has its own memory select output pin from the EBIU. Also, each of the asynchronous memory regions can be independently programmed to support burst mode or page mode flash memories.

The next region is reserved memory space. References to this region do not generate external bus transactions. Writes have no effect on external memory values, and reads return undefined values. When either of the DMAC0, DMAC1, or the USB DMA controllers address this region, the EBIU sends an error response on the internal buses to the controllers. The EBIU generates the hardware error (HWE) interrupt to the core when it is requested to access this reserved off-chip memory space.

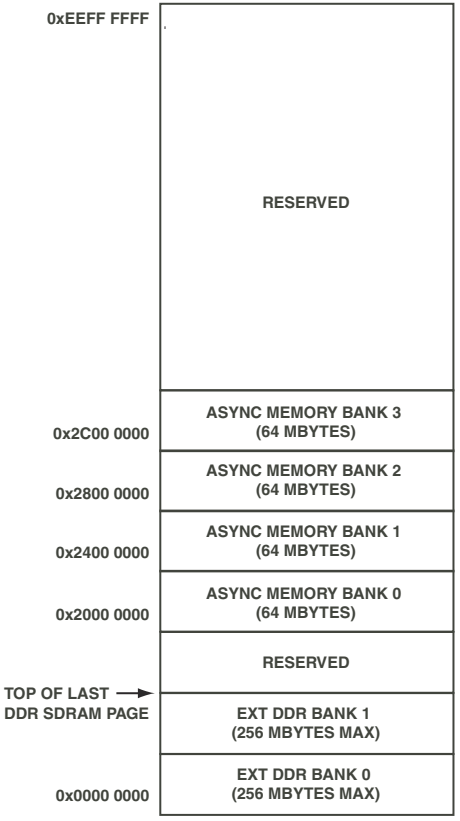


Figure 5-1. External Memory Map

Block Diagram

Figure 5-2 shows a conceptual block diagram of the EBIU. Note that the pins for the synchronous DDR memory interface are dedicated, whereas pins for the asynchronous memories are shared.

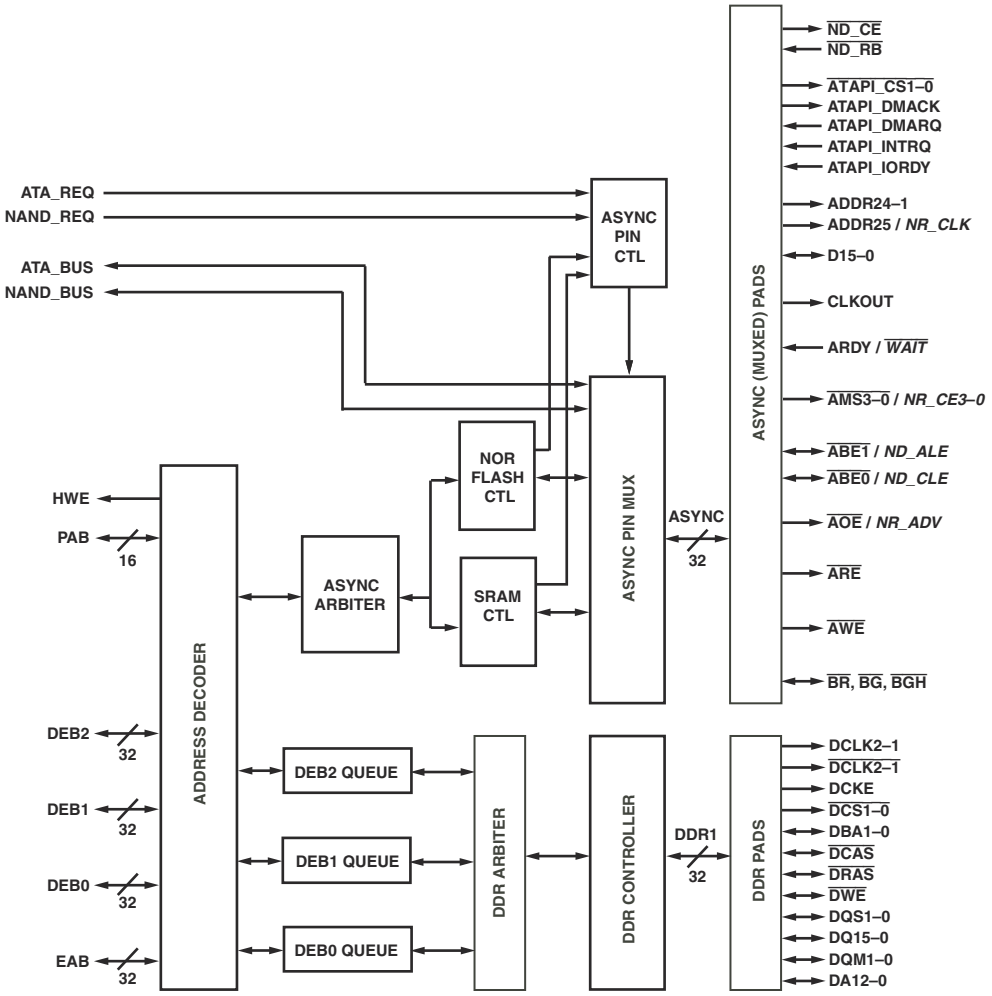


Figure 5-2. External Bus Interface Unit (EBIU) Diagram

The EBIU allows the on-chip NAND flash controller and ATAPI controller to share its asynchronous interface pins. An asynchronous pin control module (APCM) in the EBIU automatically controls the accesses to the asynchronous memory interface pins, based on requests from the ASYNC, NAND, and ATAPI with a set priority. No extra configuration is needed. The multiplexing scheme of the shared pins is summarized in Table 5-1. When reading Table 5-1, note that an "x" indicates that the pin is used by the interface, a "–" indicates that the pin is not used by the interface, and an alternate pin name indicates that the pin is used for an alternate function by the interface.

Table 5-1. EBIU Pin List (With Multiplexing)

Pins	ASYNC	FLASH	NAND FLASH	ATAPI	DDR
ADDR24–1	x	x	–	x[1]	–
ADDR25	x	NR_CLK	–	-	–
D15–0	x	x	x	x	–
$\overline{\text{AMS3-0}}$	x	NR_CE3–0	–	–	–
$\overline{\text{ABE0}}$	x	–	ND_CLE	–	–
$\overline{\text{ABE1}}$	x	–	ND_ALE	–	–
$\overline{\text{AOE}}$	x	NR_ADV	–	–	–
$\overline{\text{ARE}}$	x	x	x	-	–
$\overline{\text{AWE}}$	x	x	x	-	–
ARDY	x	$\overline{\text{WAIT}}$	–	–	–
CLKOUT	x	–	–	–	–
$\overline{\text{ND\_CE}}$	–	–	x	–	–
$\overline{\text{ND\_RB}}$	–	–	x	–	–
$\overline{\text{ATAPI\_CS1-0}}$	–	–	–	x	–
ATAPI_DMACK	–	–	–	x	–
ATAPI_INTR	–	–	–	x	–

Table 5-1. EBIU Pin List (With Multiplexing) (Cont'd)

Pins	ASYNC	FLASH	NAND FLASH	ATAPI	DDR
ATAPI_DMARQ	–	–	–	x	–
ATAPI_IORDY	–	–	–	x	–
$\overline{\text{BR}}$	x	–	–	–	–
$\overline{\text{BG}}$	x	–	–	–	–
$\overline{\text{BGH}}$	x	–	–	–	–
$\overline{\text{DCLK2-1}}$	–	–	–	–	x
DCKE	–	–	–	–	x
$\overline{\text{DCS1-0}}$	–	–	–	–	x
DBA1–0	–	–	–	–	x
$\overline{\text{DCAS}}$	–	–	–	–	x
$\overline{\text{DRAS}}$	–	–	–	–	x
$\overline{\text{DWE}}$	–	–	–	–	x
DQS1–0	–	–	–	–	x
DQ15–0	–	–	–	–	x
DQM1–0	–	–	–	–	x
DA12–0	–	–	–	–	x

1 Note that some of the pins listed in Table 6-1 are multiplexed with GPIO, especially the address lines ADDR4–ADDR25. Set the general purpose port multiplexing before using them as asynchronous memory interface, NAND flash interface, or ATAPI interface. For more information see Chapter 9, "General-Purpose Ports".

On-Chip System Interfaces

The EBIU functions as a slave on five buses internal to the ADSP-BF54x processor processor, as follows:

- A 32-bit external access bus (EAB), mastered by the core, for external memory access
- A 16-bit DMA external bus (DEB0), mastered by DMA controller1, in response to external memory access requests from any DMAC0 (16-bit) channel
- A 32-bit DMA external bus (DEB1), mastered by DMA controller2, in response to external memory access request from any DMAC1 (32-bit) channel
- A 32-bit DMA external bus (DEB2), mastered by the DMA controller in the USB module
- A 16-bit PAB bus, mastered by the core, to access the system memory-mapped registers (SMMR) in the EBIU

These are synchronous interfaces, clocked by SCLK. The EAB, DEB0, DEB1, and DEB2 (USB) provide access to both synchronous DDR SDRAM and asynchronous external memories, including page mode and burst mode NOR flash memories.

Error Detection

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

- Any access to the reserved off-chip memory space
- Any access to disabled external memory bank
- Any access to an unpopulated area of a DDR SDRAM memory bank

If the core requested the faulting bus operation, the bus error response from the EBIU is routed to the HWE interrupt internal to the core. If the DMA master issues the request for the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core. In both cases, the error address is latched in the corresponding EBIU error address register. The EBIU continues to assert the error response until explicitly cleared. The interrupt handler must write a 1 to the corresponding bit(s) in the `EBIU_ERRMST` register to clear the error condition (HWE). If the nested interrupt feature is enabled in the `SYSCFG` register (by setting the `SNEN` bit), then bit(s) in the `EBIU_ERRMST` register must be cleared at the beginning of the interrupt handler routine. Note that this behavior is specific to the ADSP-BF54x processor product.

System Arbitration

As mentioned earlier, the EBIU implements two different memory interfaces that provide simultaneous accesses to DDR SDRAM and asynchronous memory in response to requests on any of the four internal data access buses. For example, while the DDR controller services a core request to DDR SDRAM memory, the ASYNC could service a DMA

request to asynchronous or flash memory. Although the synchronous and asynchronous memories run at different speeds, the EBIU ensures that data is returned to the requestor in the correct order.

To take advantage of the high performance DDR interface and the independent asynchronous memory interface, and to maintain correct order of data transfers on the internal buses, the EBIU implements some arbitration modules that augment the DDR controller and the ASYNC memory.

Address Resolution

The EBIU address decoder block accepts the commands (read/writes) from the EAB and DMA buses (DEB0, DEB1, and DEB2). It then processes them and transfers them to the DDR queue manager (QM) block or the asynchronous memory controller block based on the address being accessed.

If the address happens to be in the reserved region (based on the memory configuration), it generates accordingly the required number of acknowledgements along with the error signal.

Reorder Unit

Because of simultaneous support of varying speed interfaces, there is a reorder engine in the EBIU for each of the system buses (DEB0, DEB1, DEB2, and EAB). The reorder engine handles out-of-order responses and makes sure that all responses from the interfaces (DDR SDRAM, asynchronous SRAM/flash) are still in the same order in which they were accepted and issued. For all read accesses, it keeps track of the states of all the requests that went to the EBIU controllers and makes sure that the responses are sent back to the original requestors in order. For write requests, each queue maintains the order in which the responses were transferred with the bus.

The following example shows out-of-order execution between the DDR interface and the ASYNC interface.

The order in which the requests are accepted and issued to the controllers is as follows:

- Cycle 1: ASYNC Read Request-1
- Cycle 2: DDR Read Request-1
- Cycle 3: DDR Read Request-2
- Cycle 4: DDR Read Request-3
- Cycle 5: DDR Read Request-4

Since the DDR interface is much faster than the ASYNC interface, the DDR read data will be available from the DDR QM block much earlier than the ASYNC interface. So the reorder engine instructs the DDR QM to stop giving the read data and hold it until the ASYNC read data is available.

- Cycle 4: DDR Read Data-1 is available but is blocked and stored in DDR QM block
- Cycle 5: DDR Read Data-2 is available but is blocked and stored in DDR QM block
- Cycle 6: ASYNC Read Data-1 is available and DDR Read Data-3 is available
- Only ASYNC Read Data-1 is now passed on to the system bus
- Cycle 7: DDR Read Data-1 is passed on to the system bus
- Cycle 8: DDR Read Data-2 is passed on to the system bus
- Cycle 9: DDR Read Data-3 is passed on to the system bus

The first access request from the system bus to the ASYNC is issued immediately (same cycle). Subsequent requests are issued only when the first access request is completed. Two consecutive requests to the ASYNC block the next access (any, including DDR access from that bus that initiated the accesses). However, accesses to DDR from other buses are not blocked.

DDR Queue Manager

To optimize for the high throughput of the DDR interface, the EBIU implements three identical queue modules for each of the DEB buses. The queue managers perform the following functions:

- Enable peripherals to utilize higher throughput provided by DDR SDRAM
- Optimize requests to the DDR controller to achieve maximum utilization of the DDR memory bus
- Handle data coherency between the DEB and core buses

DDR Arbitration

The DDR arbiter handles requests from all four system interface buses (DEB0, DEB1, DEB2, and EAB) and prefetches requests from all the DEB queue blocks. The arbiter has a fixed priority as shown in the following:

1. Core TESTSET instruction (highest)
2. Forced write access (by DEB queue manager)
3. Urgent DMA access
4. Core access

5. Normal DMA read access through DEB queue manager
6. Normal DMA write access through DEB queue manager
7. Prefetch buffer access (lowest)

Note, there is a further programmable priority scheme for the three DEB buses when DMA wins arbitration (urgent or normal access). The arbitration priority between the DEB buses are determined by bits [10:8] of the DDR queue configuration register (`EBIU_DDRQUE`) as follows:

- 000:DEB0>DEB1>DEB2 (*default*)
- 001:DEB1>DEB0>DEB2
- 010:DEB2>DEB0>DEB1

Table 5-2 summarizes the arbitration scheme, in DDR SDRAM memory interface.

Table 5-2. DDR Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	**DEB_ARB_PRIORITY: 001 (1>0>2)**	**DEB_ARB_PRIORITY: 010 (2>0>1)**
Core TESTSET	Core TESTSET	Core TESTSET
Forced DEB Writes DEB0 WRITE DEB1 WRITE DEB2 WRITE	Forced DEB Writes DEB1 WRITE DEB0 WRITE DEB2 WRITE	Forced DEB Writes DEB2 WRITE DEB0 WRITE DEB1 WRITE
Urgent DMA DEB0 READ DEB1 READ DEB2 READ DEB0 WRITE DEB1 WRITE DEB2 WRITE	Urgent DMA DEB1 READ DEB0 READ DEB2 READ DEB1 WRITE DEB0 WRITE DEB2 WRITE	Urgent DMA DEB2 READ DEB0 READ DEB1 READ DEB2 WRITE DEB0 WRITE DEB1 WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE

Table 5-2. DDR Arbiter Priority Scheme (Cont'd)

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Normal DMA READ DEB0 READ DEB1 READ DEB2 READ	Normal DMA READ DEB1 READ DEB0 READ DEB2 READ	Normal DMA READ DEB2 READ DEB0 READ DEB1 READ
Normal DMA WRITE DEB0 READ DEB1 READ DEB2 READ	Normal DMA WRITE DEB1 READ DEB0 READ DEB2 READ	Normal DMA WRITE DEB2 READ DEB0 READ DEB1 READ
Prefetch Access DEB0 READ DEB1 READ DEB2 READ	Prefetch Access DEB1 READ DEB0 READ DEB2 READ	Prefetch Access DEB2 READ DEB0 READ DEB1 READ

The EBIU adds further control to the DDR arbitration by allowing a normal DMA access to be elevated to urgent DMA access by setting bits [14:12] in the DDR queue configuration register (EBIU_DDRQUE) as follows:

```
Bit[12] = 1 : DEB0 Normal DMA treated as Urgent
          0 : DEB0 Normal DMA treated as Normal (Default)

Bit[13] = 1 : DEB1 Normal DMA treated as Urgent
          0 : DEB1 Normal DMA treated as Normal (Default)

Bit[14] = 1 : DEB2 Normal DMA treated as Urgent
          0 : DEB2 Normal DMA treated as Normal (Default)
```

Table 5-3 summarizes the arbitration scheme for asynchronous memory interface.

Table 5-3. ASYNC Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Core TESTSET	Core TESTSET	Core TESTSET
Urgent DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE
Normal DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE

The priority schemes described in Table 5-2 (for DDR) and Table 5-3 (for ASYNC) are from the arbiters' perspective. The priority schemes are followed by the arbiters only when they are ready to arbitrate, not when the EBIU receives requests on the DEB or processor buses. For example, a DEB bus may indicate urgent during a request, but if the urgent signal goes away before the arbiter arbitrates, the DEB request is treated as a regular request. Also note, that the DEB queue logic blocks optimize the DEB bus requests (for example, line hit, prefetch during reads, packing during writes, and others). Because of these optimizations, the DEB bus requests may not show up at the arbiters immediately and they may be in a different order.

DDR SDRAM Controller

The ADSP-BF54x processor processor is available with either a DDR SDRAM or a Mobile DDR SDRAM controller module on chip. Each of these has different specifications. Consult the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the proper nominal voltage and working voltage range for the various products in the ordering guide. See Chapter 19, “System Design” for more information. Unless specifically noted, all references to "DDR SDRAM controller" apply to both standard DDR and mobile DDR controllers.

The DDR SDRAM controller (SDC) enables a transfer of data to and from synchronous DDR SDRAM with a maximum data rate of 532M bytes per second at a clock frequency of 133 MHz using both the edges of the clock. It supports a glueless interface with two external banks, controlled by the memory chip select pins ($\overline{\text{DCS1-0}}$), of standard DDR SDRAMs of 64M bit to 512M bit with configurations x4, x8, x16 as shown in the following tables, up to a maximum total capacity of 256M bytes of SDRAM per chip select. The interface includes timing options to support additional buffers between DDR SDRAM and the EBIU to handle capacitive loads of large memory arrays.

Features

The following sections describe features of DDR SDRAM controllers.

DDR SDRAM Controller

The features of the DDR SDRAM controller (SDC) are:

- Supports industry-standard, double-data rate (DDR SDRAM) from 64M bit to 512M bit device sizes with a configuration of x4, x8, or x16
- Provides 16-bit data interface to DDR SDRAM

- Supports up to 256M bytes of DDR SDRAM with one external bank
- Supports up to two external banks
- Provides page hit detection to support multiple column accesses within the same row
- Provides eight internal row address registers to keep track of eight open rows (two chip select with four internal banks each)
- Supports fixed SDRAM burst length of two
- Provides programmable SDRAM access timing parameters
- Provides automatic refresh generation with programmable refresh intervals
- Supports self-refresh mode to reduce system power consumption

Mobile DDR SDRAM Controller

Mobile DDR is referred to in the industry as Low-Power DDR. The following mobile (Low-Power) DDR features are supported on the ADSP-BF54x processor processor mobile DDR SDRAM interface.

Partial Array Self-refresh

PASR is a new feature introduced to mobile DDR memories. PASR is a memory power-saving feature which limits the amount of memory to be refreshed. The ADSP-BF54x processor processor mobile DDR interface supports this feature (PASR bits of the MODE register).

Memory Driver Strength

Mobile DDR memories may support setting their drive strength (1/8, 1/4, 1/2 full). The ADSP-BF54x processor processor mobile DDR SDRAM interface supports configuring mobile DDR memory drive strength (DS bits of MODE register). Note, this configures the memory's drivers not the ADSP-BF54x processor processor's drivers.

Temperature Compensated self-refresh

TCSR is a new feature introduced to mobile DDR memories, that adjusts the refresh rate to match the die temperature. The ADSP-BF54x processorM mobile DDR SDRAM interface supports programming TCSR (if the memory supports it), using the TCSR bits of the MODE register.

Unsupported Mobile DDR SDRAM Controller Features

The ADSP-BF54x processorM processor mobile SDRAM controller does not support the following features which are referenced in the JEDEC specification and may be supported by the memory vendor.

Deep Power Down

Deep power down is a memory feature where the memory is placed in a low power state (the array is powered down) while the system power remains applied. The contents of the memory would be lost in this state. The ADSP-BF54x processor processor mobile DDR SDRAM controller does not support placing a memory in this state.

Clock Stop Mode

Clock stop mode is a memory feature where the memory can have some power savings in logic associated with its clock logic while not being accessed. The ADSP-BF54x processor processor mobile DDR SDRAM controller does not support Clock Stop Mode.

Clock Frequency During Operation

This is not a recommended procedure. However, if the application requires changing the clock frequency (SCLK) during operation, it can be done but only under the following conditions.

1. No memory accesses are in progress.
2. In the event SCLK frequency is being reduced, new control settings have been made before the frequency change is initiated.

DDR SDRAM Memory Interface

This is a DDR SDRAM compliant interface. None of the signals in this interface is multiplexed with any other signals on the chip. ADSP-BF54x processor products equipped with a standard DDR SDRAM controller support the standard DDR specification only. Similarly, ADSP-BF54x processor products equipped with a Mobile DDR SDRAM controller support only low power mobile DDR devices. Refer to the product data sheet for actual specifications.

Table 5-4. DDR SDRAM Memory Interface

Name	Type	Description
DCLK1 / $\overline{\text{DCLK1}}$	O	Output clock signals to DDR SDRAM chips. Use as differential clock signals to DDR SDRAM.
DCLK2 / $\overline{\text{DCLK2}}$	O	Output clock signals to DDR SDRAM chips. Use as differential clock signals to DDR SDRAM. Same as DDR_CLK1.
DCKE	O	Clock enable
$\overline{\text{DCS1-0}}$	O	Chip select: One chip select for each of the two external banks
DBA1–0	O	Chip select: One chip select for each of the two external banks
$\overline{\text{DCAS}}$	O	Column address select
$\overline{\text{DRAS}}$	O	Row address select
$\overline{\text{DWE}}$	O	Write enable

Table 5-4. DDR SDRAM Memory Interface (Cont'd)

Name	Type	Description
DQS1-0	IO	Data Strobe: output with write data, input with read data. DQS is edge aligned with read data, but centered with write data. It is generated by the DDR controller during write access.
DQ15-0	IO	DDR data input and output. DDR SDRAM has twice the data rate.
DQM1-0	IO	Data mask for writes. DM turns the out buffers off for writes. For Write, DM specifies the bytes to be written. It is also used to mask a single Write during an access cycle of burst length = 2.
DA12-0	O	Memory address bits: Indicates row and column address and signals auto-precharge. When 64M bit and 128M bit SDRAM are used, only DDR_ADDR[11:0] are used as addresses and BA [1:0] are used as bank select. When 256M bit and 512M bit DDR SDRAM are used, DDR_ADDR [12:0] are used as address and BA [1:0] are used as bank select.

DDR SDRAM Programming Model

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers (SMMRs), used to program the EBIU. This set of control registers is accessed across the peripheral access bus (PAB) of the extended core.

The control and status registers in the DDR controller include:

- Memory control register 0 (EBIU_DDRCTL0) address 0xFFC0 0A20
- Memory control register 1 (EBIU_DDRCTL1) address 0xFFC0 0A24
- Memory control register 2 (EBIU_DDRCTL2) address 0xFFC0 0A28
- Memory control register 3 (EBIU_DDRCTL3) address 0xFFC0 0A2C

- DDR queue manager configuration register (EBIU_DDRQUE) address 0xFFC0 0A30
- Error address register (EBIU_ERRADD) address 0xFFC0 0A34
- Error master register (EBIU_ERRMST) address 0xFFC0 0A38
- Reset control register (EBIU_RSTCTL) address 0xFFC0 0A3C

Access to the DDR controller registers (EBIU_DDRCTLx) can be made *only* after releasing the DDR controller soft reset bit in the reset control register (EBIU_RSTCTL) by writing a 1 to bit[0] of the register.

The EBIU_DDRCTL0, EBIU_DDRCTL1, EBIU_DDRCTL2 and EBIU_DDRCTL3 can not be written when the DDR controller is in self-refresh mode. Such an attempt causes the processor to hang.

Programs may write to the DDR control registers as long as the controller is not accessing memory devices. The controller responds to any writes to its registers after it finishes ongoing memory accesses.

The DDR control registers contain sensitive timing parameters and settings for the DDR SDRAM. Carefully program these registers with values that are in the operating range of the DDR being used. In addition to meeting the timing specifications defined in the DDR memory datasheet, the user must ensure that the DDR controller is configured such that tRC <=RP + tRAS.

Values in the reserved fields in these registers must be maintained according to the specification. Writing to reserved fields or writing reserved values to register bits causes incorrect function.

The programmer must not change the prefetch length fields of the EBIU_DDRQUE register during an ongoing transfer on DEB buses; otherwise unpredictable behavior may occur.

Recommended Programming Sequence

In general the following order is recommended for programming the EBIU registers.

1. EBIU_DDRQUE using a read-modify-write operation
2. EBIU_RSTCTL using a read-modify-write operation. Always set bit 0 and bit 5 as appropriate for the type of DDR memory actually connected to the ADSP-BF54x processor.
3. EBIU_DDRCTLx in any order

Out of reset/boot, by default, the ADSP-BF54x processor processor will have a VLEV setting of "F". This is programmed at the factory in OTP factory page settings page FPS04 and loaded into the VR_CTL register during preboot. See Chapter 17, “System Reset and Booting” for more details of preboot. Since only values of "D" and "E" allowed, the “F” value is officially out-of-specification. The programmer should call the on-chip ROM function bfrom_SysControl() to program a value of either "D" or "E" in the four bit voltage level field (VLEV) within the VR_CTL register.

The Mobile DDR enable bit is cleared on reset (bit 5 of EBIU_RSTCTL) and must be re-enabled along with the EBIU_RSTCTL bit 0 enable bit. The boot kernel code in on-chip ROM normally sets this bit during preboot. However the programmer should set this bit within the application code since it will be cleared, for example, if a software system reset is issued.

The general recommendation is that the programmer should always set this bit in the application code when using Mobile DDR and should not rely on preboot since booting may not occur when events such as a software system reset are invoked.

Listing 5-1. Example Assembly Code to Set `EBIU_RSTCTL` Bit 0 and 5

```
p0.l = lo(EBIU_RSTCTL);
p0.h = hi(EBIU_RSTCTL);
r1 = w[p0];
bitset (r1,0);
bitset (r1,5);
w[p0] = r1;
sync;
```

Listing 5-2. Example C Code for Call to `bfrom_SysControl()`

Only the `SYSCTRL_VRCTL` flag is required to set VLEV (see information above); but this function demonstrates setting the PLL as well.

```
#include <bfrom.h>
#include <cdefBF548.h>
    // Set new values for PLL_CTL, PLL_DIV and VR_CTL
    mystruct.uwPllCtl = 0x1000;
    mystruct.uwPllDiv = 0x0002;
    mystruct.uwVrCtl =  0x409B;
return_code = bfrom_SysControl(SYSCTRL_WRITE|SYSCTRL_PLLCTL|
SYSCTRL_PLLDIV|SYSCTRL_VRCTL|SYSCTRL_INTVOLTAGE,&mystruct,NULL);
if (return_code)
return FAIL;
```

Details of preboot, callable ROM functions, and the registers cited above can be found in Chapter 18, “Dynamic Power Management” and Chapter 17, “System Reset and Booting”.

DDR Registers

This section provides descriptions of the EBIU's memory-mapped registers (MMRs) for DDR programming.

This section describes the following registers:

"Memory Control Register 0 (EBIU_DDRCTL0)" on page 5-24
"Memory Control Register 1 (EBIU_DDRCTL1)" on page 5-25
"Memory Control Register 2 (EBIU_DDRCTL2)" on page 5-26
"Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices" on page 5-27
"Memory Control Register 3 (EBIU_DDRCTL3), Mobile DDR Devices" on page 5-28
"Error Master Register (EBIU_ERRMST)" on page 5-31
"Error Address Register (EBIU_ERRADD)" on page 5-32
"Reset Control Register (EBIU_RSTCTL)" on page 5-30

Memory Control Register 0 (EBIU_DDRCTL0)

Access to this register can be made *only* after releasing bit[0] of the `EBIU_RSTCTL` register. This register can not be written during self-refresh mode.

In addition to meeting the timing specifications defined in the DDR memory datasheet, the user must ensure that the DDR controller is configured such that `tRC <=RP + tRAS`.

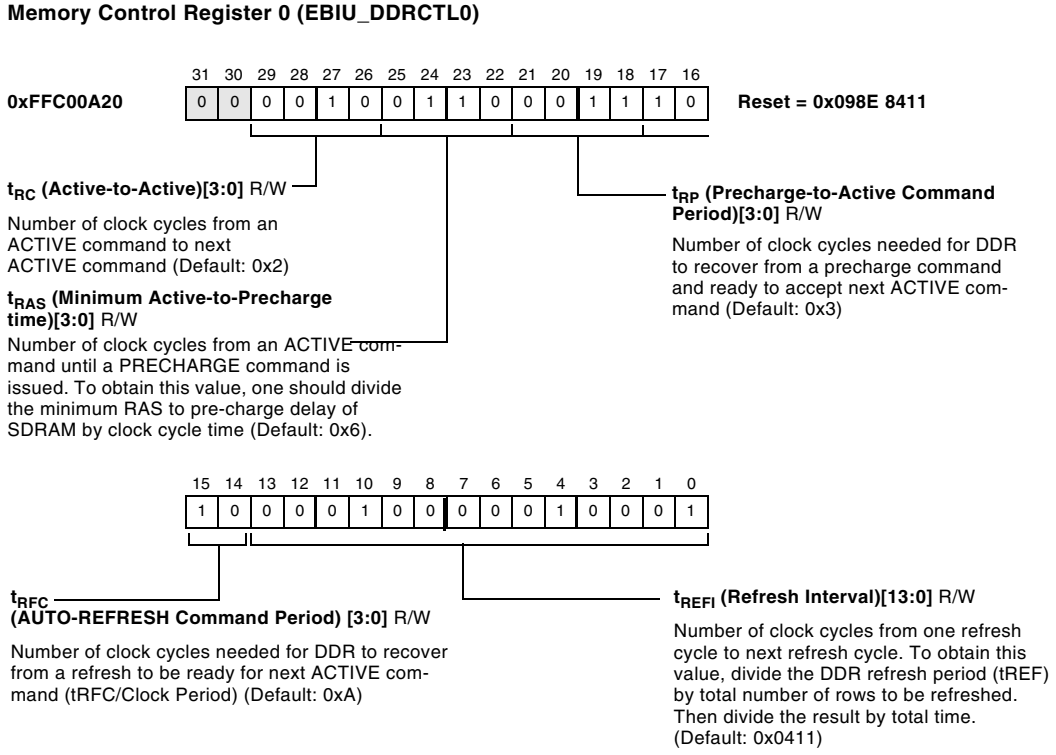


Figure 5-3. Memory Control Register 0

Memory Control Register 1 (EBIU_DDRCTL1)

Access to this register can be made *only* after releasing bit[0] of the EBIU_RSTCTL register. This register can not be written during self-refresh mode.

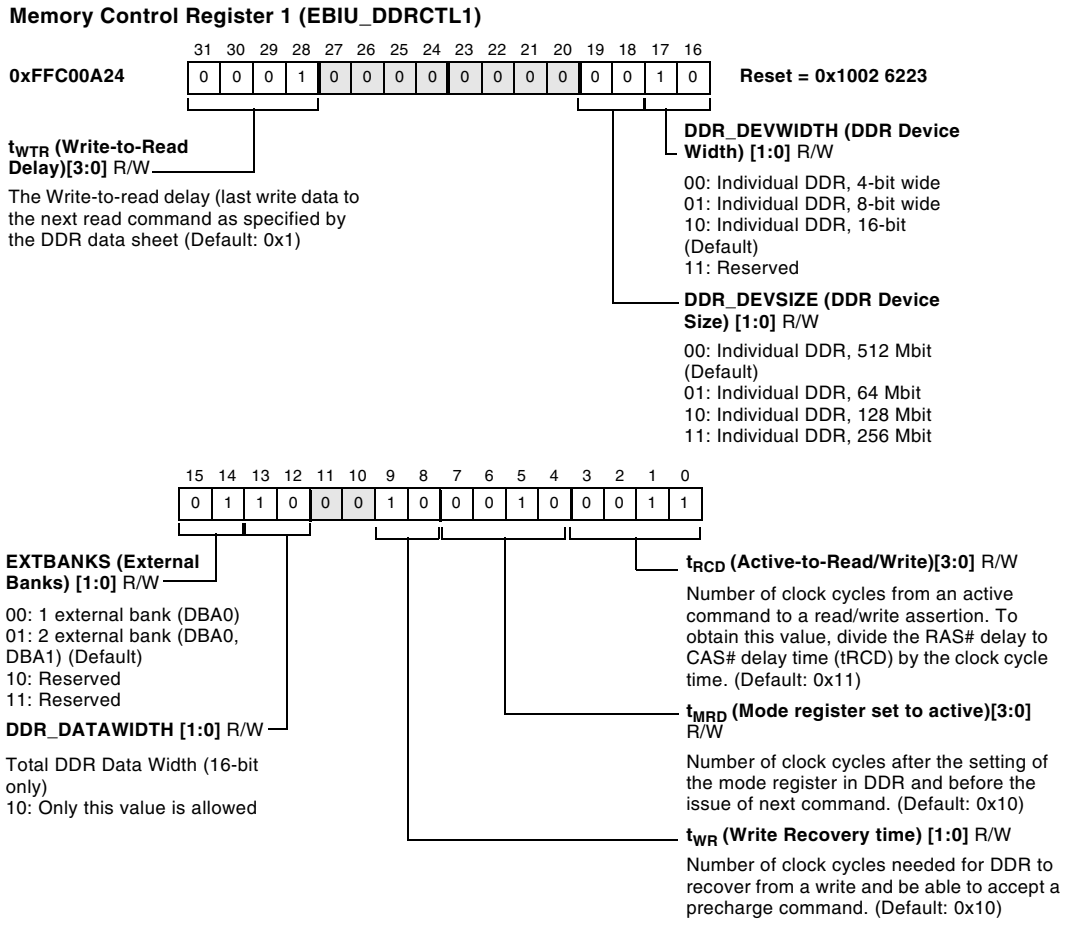


Figure 5-4. Memory Control Register 1

Memory Control Register 2 (EBIU_DDRCTL2)

Access to this register can be made *only* after releasing bit[0] of the EBIU_RSTCTL register. This register can not be written during self-refresh mode.

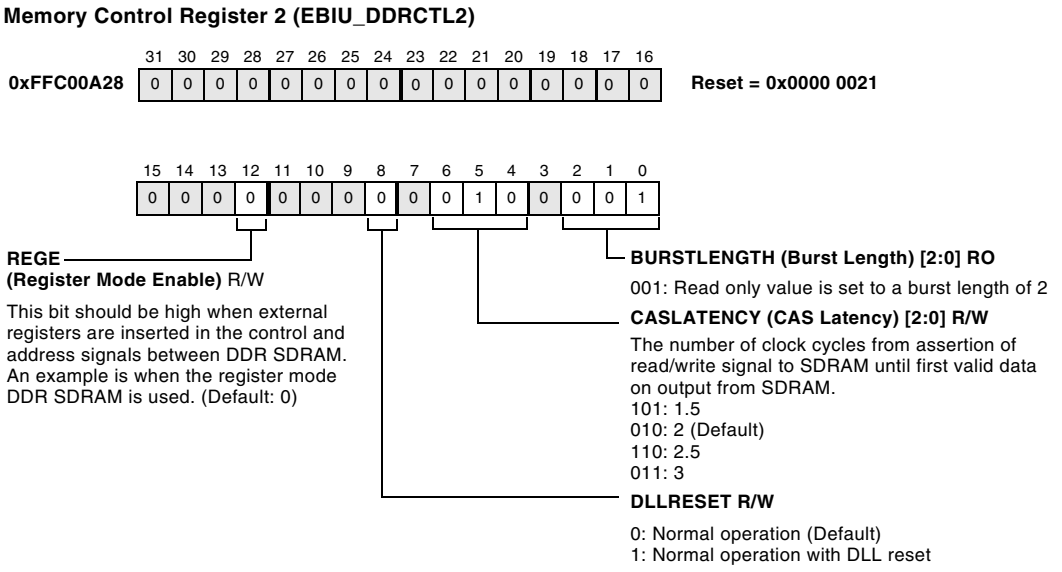


Figure 5-5. Memory Control Register 2

Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices

Access to this register can be made *only* after releasing bit[0] of the EBIU_RSTCTL register. This register can not be written during self-refresh mode.

This register is used to control (update) the content of the Extended Mode Register of a DDR memory device. The fields and bits of this register correspond directly to those of the Extended Mode Register of a DDR memory device. The programmer can update the values of the Extended Mode Register by writing to this register according to the memory vendor specification. Only values supported by the memory device should be written.

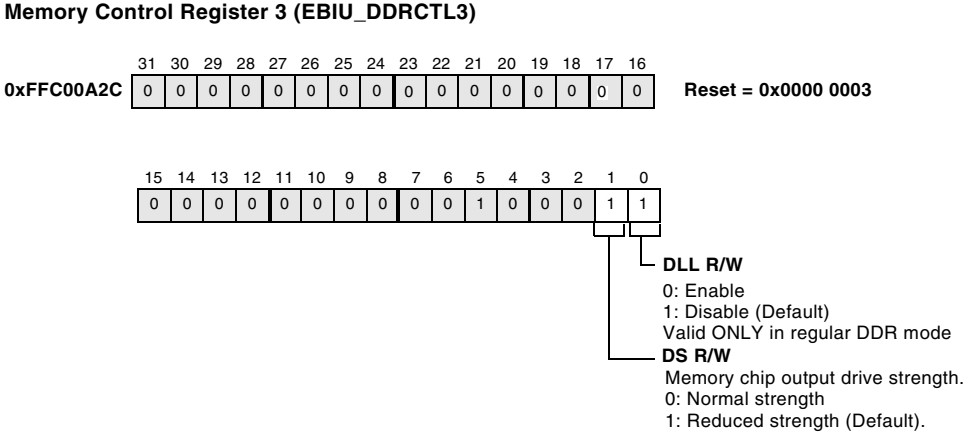


Figure 5-6. Memory Control Register 3 (Regular DDR Devices)

Memory Control Register 3 (EBIU_DDRCTL3), Mobile DDR Devices

This register is used to control (update) the content of the Extended Mode Register of a Mobile DDR memory device. The fields and bits of this register correspond directly to those of the Extended Mode Register of a Mobile DDR memory device. The programmer can update the values of the Extended Mode Register by writing to this register according to the memory vendor specification. Only valid values supported by the memory device should be written.

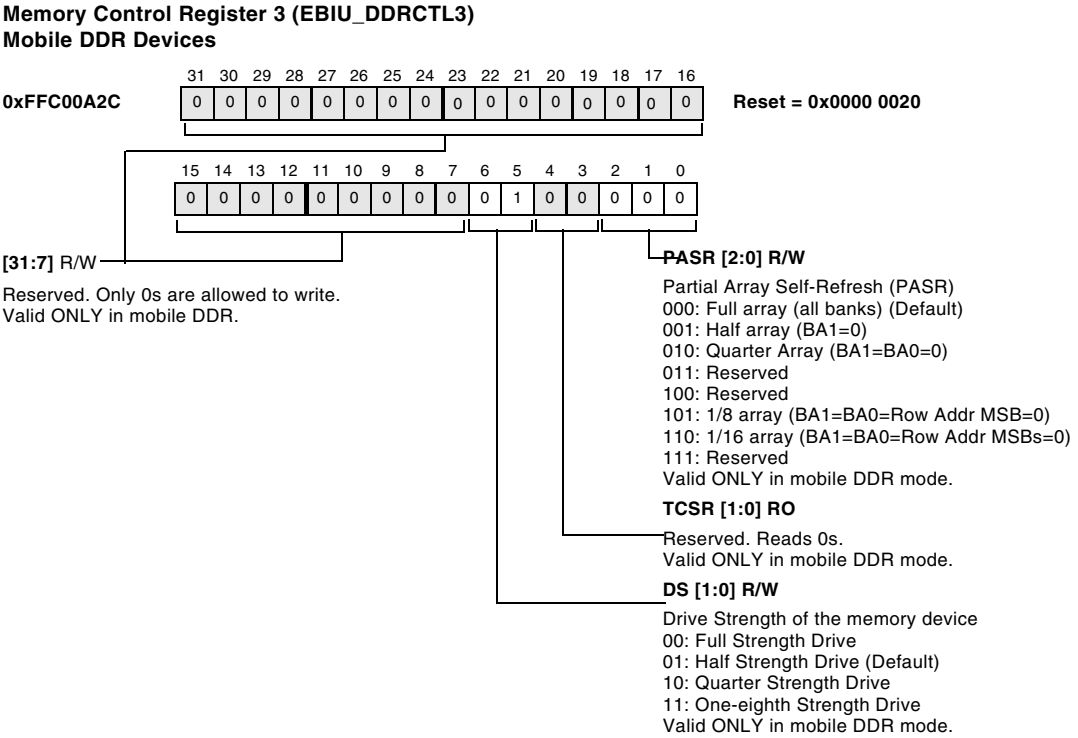


Figure 5-7. Memory Control Register 3 (Mobile DDR Devices)

Queue Configuration Register (EBIU_DDRQUE)

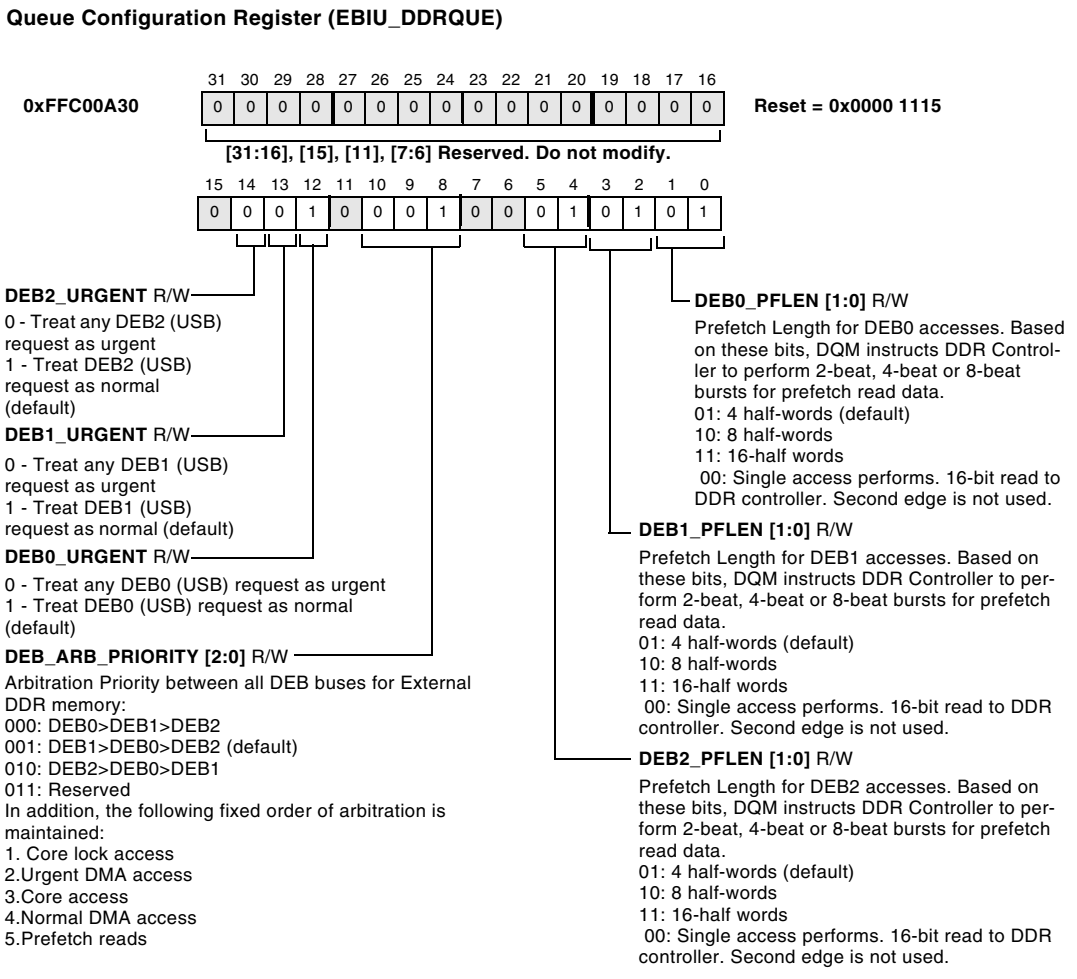


Figure 5-8. Queue Configuration Register

Reset Control Register (EBIU_RSTCTL)

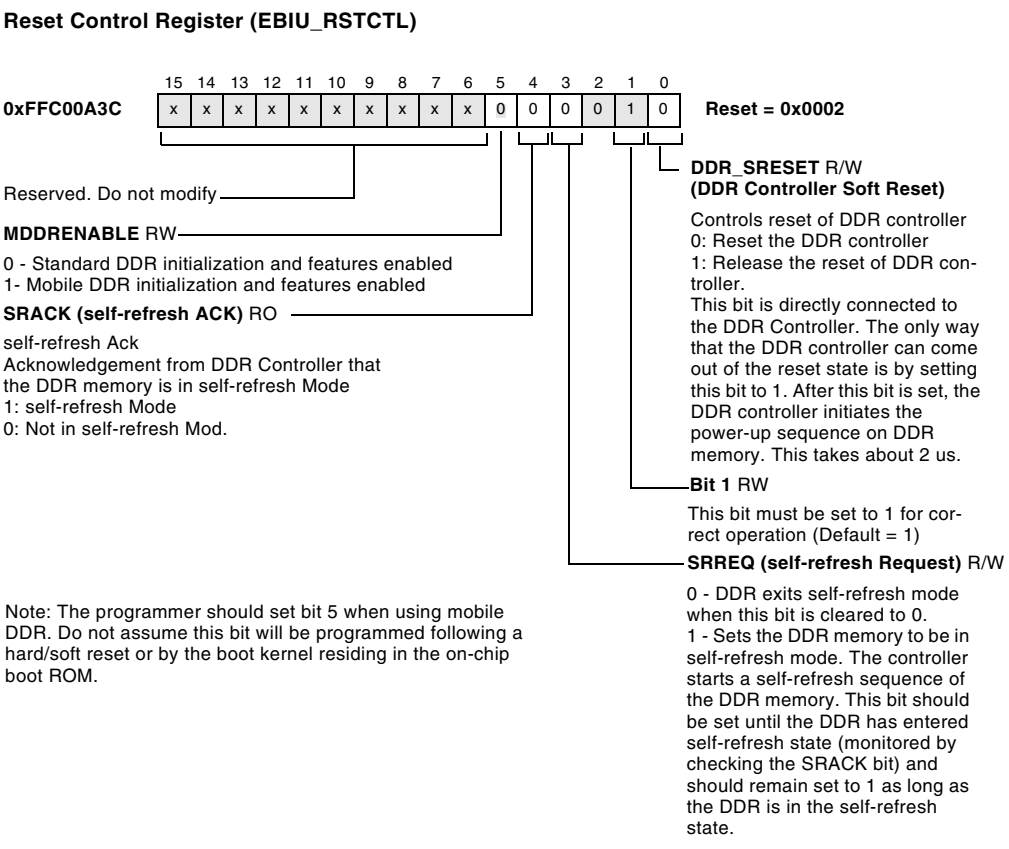


Figure 5-9. Reset Control Register 0

Error Master Register (EBIU_ERRMST)

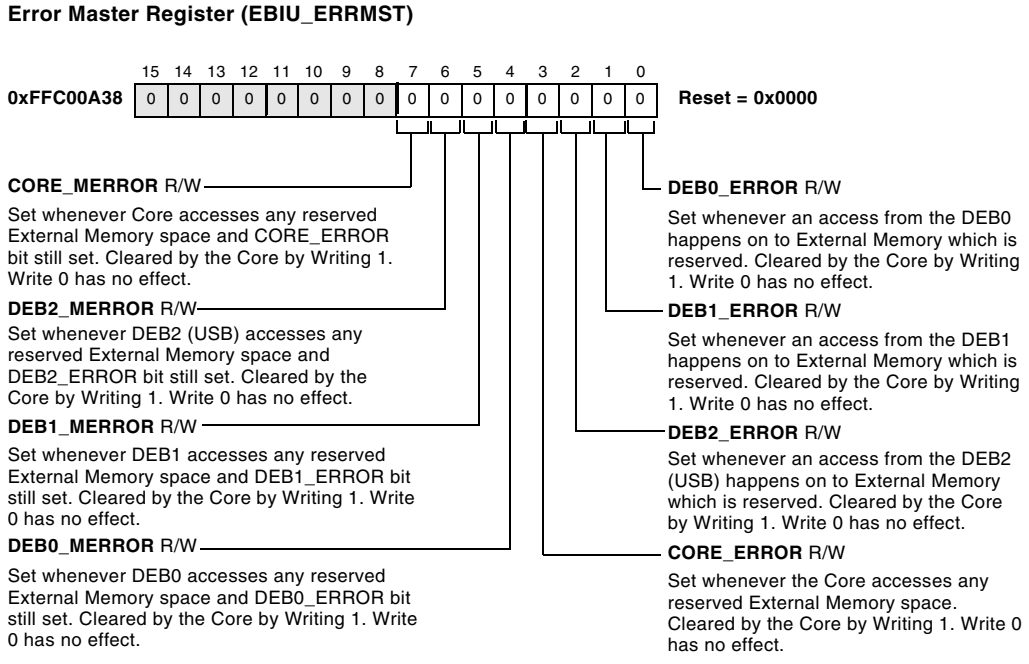


Figure 5-10. Error Master Register

Error Address Register (EBIU_ERRADD)

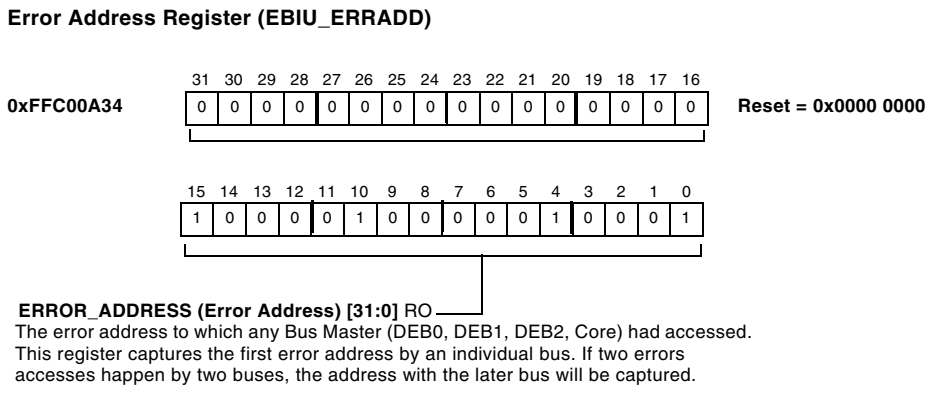


Figure 5-11. Error Address Register

Mode of Operation - DDR

The DDR SDRAM controller performs the DDR SDRAM read and write accesses based on external SDRAM memory requests by the processor core EAB, DEB0, DEB1, and DEB2 buses.

The DDR SDRAM timing, such as row and column latency, precharge timing, and row access time are programmed to default values at system reset. They also can be programmed during runtime if the application wishes to optimize the system performance. The internal counters in the DDR controller handle all the timing parameters.

Data between the DDR SDRAM controller and the DDR SDRAM device transfers at both the rising edge and falling edge of clock. The DDR SDRAM controller has the built-in data path to handle all data generation and sampling tasks.

Data Flow for 16-bit DDR SDRAMs

For read access, the DDR SDRAM drives 16 bits of data at both edges of the SDRAM clock. The DQS strobe is sampled by the DDR controller data path (synchronized with internal clock) and transferred to the DDR arbiter as a single 32-bit data. The DDR arbiter transfers the 32-bit data to the corresponding queues for which the read request command is accepted. The queue in turn transfers the same on to DMA buses or unpacks the 32-bit data word into two single half-words (16-bit) or 4 single bytes (8-bit), depending upon the DMA data width, before transferring them on to DMA buses. In the case of 32-bit wide DMA transfers, no unpacking is done.

For a core read request, the DDR arbiter transfers the 32-bit data to the core.

For write accesses, each DEB queue accepts byte/half-word/word requests from the corresponding DMA bus and packs into a 32-bit DDR SDRAM data word. A write request to the DDR arbiter is then made. The DDR arbiter then accepts a 32-bit write requests from DEB queues and the core bus, arbitrates based on arbitration priority and transfers one of the write request on to DDR controller. The DDR controller in turn writes as two 16-bit half-words on both edges of the clock (DQS strobe).

For write requests from the core, write commands are sent directly to the DDR arbiter without any packing.

The DDR SDRAM controller supports SDRAM devices of sizes of 64, 128, 256, 512 Mbits. For all device sizes it supports configurations of x4, x8 and x16 data width per SDRAM. The programmer can use multiple SDRAM devices to build a SDRAM data width of 16-bits. Both the device size and SDRAM data word size is programmable by the programmer.

The DDR SDRAM controller supports an open page policy. Open page policy takes advantage of the fact that once a row is activated, multiple accesses can be made to the same row (page) without precharging the bank.

The pipeline feature of the DDR controller and the queuing feature of the queue manager block consecutive page hit write or consecutive page hit read to/from DDR without any idle cycles between accesses.

Definition of Standard DDR Terms

The following are definitions used in the rest of this chapter.

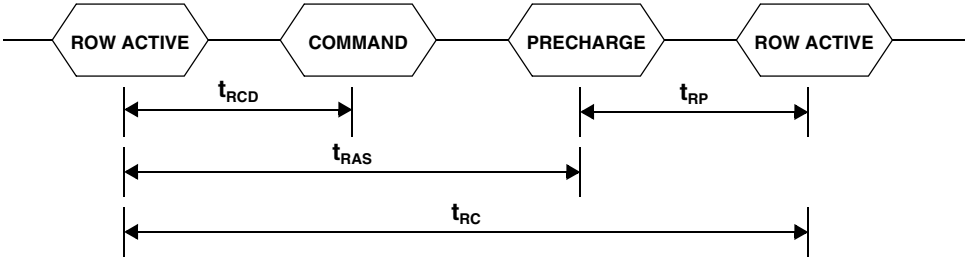


Figure 5-12. DDR Terms

Active command The active command is used to open (or activate) a row in a particular bank for subsequent access. The value on the DBA1-0 pins selects the DDR's internal bank, and the address on the DA12-0 pins selects the row. The open row is also referred to as the open page. This row (page) remains open for accesses until a precharge command is issued to that bank. In a particular bank, only one row can be open at any time. A precharge command must be issued before opening a different row in the same bank.

Precharge command The precharge command is used to close (or deactivate) the open row in a particular bank or the open row in all internal banks. Once a bank is precharged, it is in an idle state and must be activated prior to any read or write commands being issued to the same bank.

Read command Precharge not supported

Write command Precharge not supported

Auto refresh command DDR data must be refreshed within a certain interval to prevent loss of data. The DDR controller automatically issues an auto refresh command to the DDR SDRAM device to refresh its memory. The DDR controller refreshes one row each time.

The refresh interval is programmable by the programmer in the control register. All control registers can be programmed during runtime. The refresh interval field in the control register measures the interval for auto-refresh in terms of clock cycles. If the cycle time of the system clock is 10ns, the refresh interval value should be 780 to indicate a 7.8 µs refresh interval.

The DDR SDRAM controller has an internal counter to count the refresh period. When the counter expires, the controller precharges all the banks and then issues an auto-refresh command to the SDRAM—if the SDRAM is in idle state. If the SDRAM is being accessed for read, write or other commands, the precharge and auto refresh commands are delayed until the current command is completed.

A request for an access to DDR SDRAM while auto refresh is in progress will be delayed till auto refresh is completed.

Enter Self-Refresh Mode To minimize power consumption, the DDR SDRAM controller enters the self-refresh mode under programmer control. When the SRREQ bit is set (in the EBIU_RSTCTL register), it starts the self-refresh sequence. This enables the SDRAM to continue to refresh its memory array while minimizing power consumption, resulting in no data loss. Once the SRREQ bit is set, it should not be cleared until the DDR SDRAM enters a self-refresh state, indicated by SRACK = 1 in the reset control register. The processor or DMA should not issue any further commands until the SRACK bit is set.

The DDR controller brings the DDR SDRAM to self-refresh mode by issuing self-refresh and de-asserts the DCKE signal. The DCKE signal is kept low until the DDR exits self-refresh mode.

Once in self-refresh mode, the programmer must not attempt to access the DDR memory. The programmer must also not attempt to write EBIU_DDRCTL0, EBIU_DDRCTL1, EBIU_DDRCTL2 and EBIU_DDRCTL3 registers while in self-refresh mode. *Such actions will cause the processor to hang.*

Exit from Self-Refresh Mode To exit from self-refresh mode, the SRREQ bit must be de-asserted by the programmer. The controller asserts the DCKE signal and then issues an auto-refresh after waiting for 16 clock cycles. However, DDR SDRAM devices are required to wait for 200 clock cycles before processing any read/write request. The DDR SDRAM controller keeps the SRACK bit asserted high for 200 cycles after the DCKE is asserted. After the SRACK is cleared, SDRAM is operational again and the programmer can issue normal SDRAM requests. The processor should check for SRACK being cleared and then issue any commands.

The programmer must follow the procedures described above for entering and exiting the self-refresh mode. After exiting self-refresh mode, the programmer must set the EBIU_DDRCTL0, EBIU_DDRCTL1, EBIU_DDRCTL2 and EBIU_DDRCTL3 registers appropriately for the frequency of operation and DDR memory specification.

Returning from Hibernate After putting the DDR memory in self-refresh mode, it is common for the Blackfin processor to go to the Hibernate state, with no power applied to itself. Later, when the Blackfin processor powers up and returns from the Hibernate state, the DDR will enter the self-refresh mode. Enabling the DDR controller will cause the DDR memory to exit self-refresh mode.

Once the DDR controller is enabled, it goes through the normal initialization sequence and wakes up the DDR memory from self-refresh mode to its normal mode of operation. When the DDR controller is enabled (after returning from hibernate), the programmer must, as always, set the EBIU_DDRCTL0, EBIU_DDRCTL1, EBIU_DDRCTL2 and EBIU_DDRCTL3 registers appropriately for the frequency of operation and DDR memory specification.

Mode Register Set The mode register is the DDR internal configuration register containing programmer defined parameters. The mode register set command is issued by the DDR controller automatically during power on initialization and when the programmer writes to EBIU_DDRCTL2.

Extended Mode Register Set The extended mode register set command is issued by the DDR controller automatically during power on initialization and when the programmer writes to EBIU_DDRCTL3. Extended mode register set and mode register set differ by the encoding of the DBA1-0 signals.

Table 5-5. DDR SDRAM Commands

CS#	RAS#	CAS#	WE#	BA[1:0]	Commands
L	L	L	L	00	Mode register set
L	L	L	L	01	Extended mode register set
H	X	X	X	X	Command inhibit (NOP)
L	L	H	H	X	Active
L	H	L	H	X	Read
L	H	L	L	X	Write
L	L	H	L	X	Precharge
L	L	L	H	X	Refresh
L	L	L	L	X	Mode register set/extended mode register set
L	H	H	L	X	Burst terminate
-	-	-	-	L	Write enable/output enable
-	-	-	-	H	Write inhibit/output high -Z

Burst Length The burst length determines the number of words the DDR stores or delivers after detecting a single write or read command, respectively. The burst length is programmed in the SDRAM mode register during the power-up sequence. The DDR controller, for the ADSP-BF54x processor processor, only supports burst length = 2 mode.

Burst Stop Command The burst stop command is one of several ways to terminate a burst read or write operation. Since the SDRAM burst length is always programmed to be 2, the DDR controller does not need any burst stop command.

Burst Type The burst type determines the access order in which the DDR delivers burst data after detecting a read command or stores burst data after detecting a write command. The burst type is programmed in the DDR mode register during the power-up sequence. Burst type can be sequential or interleaved. Since the DDR controller only supports a burst length of 2, the burst type does not matter. The ADSP-BF54x processor processor 's DDR controller always sets the burst type to sequential-accesses-only during the SDRAM power-up sequence.

CAS Latency (also t_{AA}, t_{CAC}, t_{CL}). The column address strobe ($\overline{\texttt{DCAS}}$) latency is the delay, in clock cycles, between when the SDRAM detects the read command and when it provides the data at its output pins. The $\overline{\texttt{DCAS}}$ latency is programmed in the SDRAM mode register during the power-up sequence. The speed grade of the device and the application's clock frequency determine the value of the $\overline{\texttt{DCAS}}$ latency. The DDR controller supports $\overline{\texttt{DCAS}}$ latencies of 1.5, 2, 2.5, and 3 clocks.

CBR (CAS before RAS) Auto-Refresh When the DDR controller refresh counter times out, it precharges all four banks of SDRAM and then issues an auto-refresh command to them. This causes the SDRAMs to generate an internal CBR refresh cycle. When the internal refresh completes, all four DDR internal banks are precharged.

DQM Data I/O Mask Function The DQM1-0 pins provide a byte masking capability on 8-bit writes to DDR. The DQM1-0 pins are not used to mask data on read cycles.

Internal Bank In a DDR, there are several internal memory banks. These banks are selected by the bank address (DBA1-0) pins.

Page Hit Detection The DDR controller stores the row address in the row address register each time it activates a bank. Internally the DDR controller has four row address registers, one for each bank. Once a bank is activated for read or write, the bank remains active. When a new access request arrives to the DDR SDRAM controller, it automatically checks the internal row address register. If the new access is for the same row (page hit), the DDR SDRAM controller skips the active command and directly issues the read/write command to access the DDR.

Maximum Bank Active Time Each DDR bank can remain in an active state up to hundreds of microseconds, but it must be precharged again before the maximum active–to–precharge time is exceeded. The DDR SDRAM controller assures that each bank does not exceed the maximum active-to-precharge time by use of a refresh interval. Since the refresh period is smaller than the maximum active–to–precharge time in SDRAMs and all banks must be idle before a refresh can be issued, no bank will remain in the active state for more than the active-to-pre-charge time. For each refresh issued, the DDR SDRAM controller checks that all banks are idle. If any bank is active, the controller issues an all bank precharge command to DDR before the refresh command.

The programmer must make sure that the refresh cycle that is programmed in `EBIU_DDRCTL0` is smaller than the active–to–precharge time.

Page Miss Access When a DDR SDRAM access generates a page miss that the bank is precharged (deactivated), the DDR controller starts the access with the ACTIVE command. If the bank is active but the row address is a mismatch, the DDR controller first issues a precharge command. After the precharge-to-active delay, the DDR controller issues the active command and then a read or write command to access the memory. If the bank is already precharged, the precharge command is skipped.

Register Mode DDR Support The DDR SDRAM controller supports DDR SDRAM systems with and without external registers for address and control signals. The buffered mode is functionally identical to using single discrete SDRAM devices. The control and address signals are buffered on the board to reduce loading to the SDRAM controller. The REGE bit must be set to 0 to support discrete and buffered mode.

Register mode is designed for systems that have external registers for each control and address signal between the DDR controller and the DDR SDRAM. The REGE bit is set to 1 to enable the register mode. When register mode is enabled, the latency of all accesses is increased by one system clock cycle.

DDR SDRAM System Organization

DDR devices are available with 4-, 8-, and16-bit data width. To build a memory system with 16-bit data, multiple x4 or x8 DDR SDRAM devices can be connected in parallel to provide the total data bits. Different data word sizes do not affect the address bit used to access the SDRAM. The word size on the interface between the DDR SDRAM controller and the DDR queue block is always double the width of the data path to the DDR because the DDR transfers two bits of data per pin per clock cycle.

All the address and control signals, with the exception of DQM1-0, are common to all SDRAM chips. The DQM1-0 signal must match with the data bits with which they are associated.

Figure 5-13 shows a DDR system of 16-bit data word made by using 512M bit (64M bytes) SDRAM devices with x8 configuration, producing 128M bytes per external memory bank.

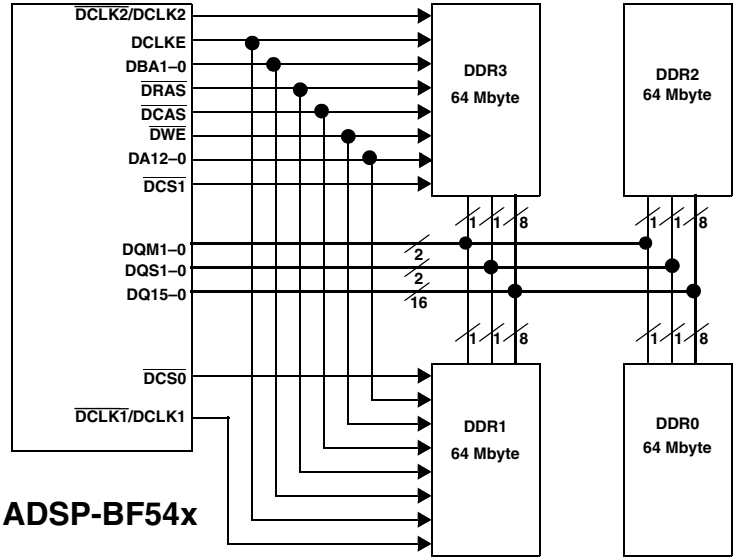


Figure 5-13. 16-bit Data Bus DDR System

DDR SDRAM Configurations Supported

The ADSP-BF54x processor DDR SDRAM controller supports different sizes of SDRAM chips from 64 Mbit to 512 Mbit. The following tables list the supported sizes.

Table 5-6. Using 64 Mbit (8M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External Bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	32M bytes	22:11	10:1	24:23	26:25
16	x8	2	16M bytes	21:10	9:1	23:22	25:24
16	x16	1	8M bytes	20:9	8:1	22:21	24:23

Table 5-7. Using 128 Mbit (16M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External Bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	64M bytes	23:12	11 :1	25:24	27:26
16	x8	2	32M bytes	22:11	10:1	24:23	26:25
16	x16	1	16M bytes	21:10	9:1	23:22	25:24

Table 5-8. Using 256 Mbit (32M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	128M bytes	24:12	11:1	26:25	28:27
16	x8	2	64 M bytes	23:11	10:1	25:24	27:26
16	x16	1	32 M bytes	22:10	9:1	24:23	26:25

Table 5-9. Using 512 Mbit (64 M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	256 M bytes	25:13	12:1	27:26	29:28
16	x8	2	128 M bytes	24:12	11:1	26:25	28:27
16	x16	1	64 M bytes	23:11	10:1	25:24	27:26

DDR Timing Parameter Definitions

ACTIVE-to-PRECHARGE Command Delay (t_{RAS})–The required delay between issuing an active command and issuing a precharge command and between the self-refresh command and the exit from self-refresh.

PRECHARGE Command Period (t_{RP})–The required delay between issuing a precharge command and issuing an activate command, between a precharge command and issuing an auto-refresh command, and between a precharge command and issuing a self-refresh command.

ACTIVE_A-to-ACTIVE_A Delay t_{RC})–The required delay between issuing successive activate commands to the same SDRAM internal bank. The programmer must ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

ACTIVE-to-READ/WRITE Delay (t_{RCD})–The required delay between an active command and the start of the first read or write command

AUTO-REFRESH Command Period (t_{RFC})–The required delay between issuing an auto-refresh command and an active command.

Average Refresh Interval (t_{REFI})–The number of cycles from one refresh command to next refresh command.

WRITE-to-READ Delay (t_{WTR})–The number of cycles between last write data and next read command.

Write Recovery Time (t_{WR})–The clock cycles needed for the SDRAM to recover from a write command and be able to accept a precharge command.

LOAD MODE REGISTER Command Cycle Time (t_{MRD})–The number of clock cycles after setting of the mode register in the DDR and before issue of next command.

DDR Metrics Control Registers

The EBIU provides a set of registers and counters to monitor performance and activities on the DDR SDRAM interface and in the DDR arbiter for accesses to DDR SDRAM memory. There are 23 counters and two control registers, each 32-bits wide, for this purpose.

The following sections describe the DDR metrics control registers:
"DDR Metrics Counter Enable (EBIU_DDRMCEN) Register" on page 5-44
"DDR Metrics Counter Clear (EBIU_DDRMCCL) Register" on page 5-47
"DDR READ Access Count (EBIU_DDRBRCx) Registers" on page 5-49
"DDR WRITE Access Count (EBIU_DDRBWCx) Registers" on page 5-50
"DDR Page ACTIVATE Count (EBIU_DDRACCT) Register" on page 5-51
"DDR TURN AROUND Count (EBIU_DDRTACT) Register" on page 5-51
"DDR AUTO-REFRESH Count (EBIU_DDRARCT) Register" on page 5-51
"DDR Grant Count (EBIU_DDRGCx) Registers" on page 5-51

DDR Metrics Counter Enable (EBIU_DDRMCEN) Register

This 32-bit system MMR independently controls different DDR metrics counters. Each bit in this register (except bits[25:24]) enables and disables the corresponding counter. When a bit is set to 1, the corresponding counter starts. When a bit is 0, the corresponding counter stops counting but is not cleared. The corresponding bit in the DDR metrics counter clear (`EBIU_DDRMCCL`) register must be set to clear the counter (see Table 5-10 and Table 5-11).

Table 5-10. DDR Metrics Counter Enable Register

Address	Register Name	Size	Reset Value
0xFFC0 0AC0	DDR Metrics Counter Enable Register	32	0x0000 0000

Table 5-11. DDR Metrics Counter Enable Register Bits

Name	Offset	Access	Description
Reserved	31:30	RO	Reads 0
GCCONTROL DDR Grant Count Control	25:24	R/W	Specifies 4 different schemes for DDR Grants Count (see Table 5-14 on page 5-52): 00: Core, DEB0, DEB1, DEB2,(Default) 01: Core, DEB0WR, DEB0RD, DEB0PF 10: Core, DEB1WR, DEB1RD, DEB1PF 11: Core, DEB2WR, DEB2RD, DEB2PF
GC3ENABLE DDR Grant Count Register 3 Enable	23	R/W	1: Enable Grant Count Register 3 0: Disable Grant Count Register 3(Default)
GC2ENABLE DDR Grant Count Register 2 Enable	22	R/W	1: Enable Grant Count Register 2 0: Disable Grant Count Register2(Default)
GC1ENABLE DDR Grant Count Register 1 Enable	21	R/W	1: Enable Grant Count Register 1 0: Disable Grant Count Register1(Default)
GC0ENABLE DDR Grant Count Register 0 Enable	20	R/W	1: Enable Grant Count Register 0 0: Disable Grant Count Register01(Default)
Reserved	19	RO	Reads 0
ARCENABLE Total DDR Auto-Refresh Count Enable	18	R/W	1: Enable Auto-Refresh Count 0: Disable Auto-Refresh Count (Default)
RWTCENABLE Total DDR R/W Turn Around Count Enable	17	R/W	1: Enable Turn Around Count 0: Disable Turn Around Count (Default)

Table 5-11. DDR Metrics Counter Enable Register Bits (Cont'd)

Name	Offset	Access	Description
ROWACTCENABLE Total DDR Row ACTIVATE Count Enable	16	R/W	1: Enable Row Activate Count 0: Disable Row Activate Count (Default)
B7RCENABLE Bank7 Read Count Enable	15	R/W	1: Enable Read Count to Bank7 0: Disable Read Count to Bank7(Default)
B6RCENABLE Bank6 Read Count Enable	14	R/W	1: Enable Read Count to Bank6 0: Disable Read Count to Bank6(Default)
B5RCENABLE Bank5 Read Count Enable	13	R/W	1: Enable Read Count to Bank5 0: Disable Read Count to Bank5(Default)
B4RCENABLE Bank4 Read Count Enable	12	R/W	1: Enable Read Count to Bank4 0: Disable Read Count to Bank4(Default)
B3RCENABLE Bank3 Read Count Enable	11	R/W	1: Enable Read Count to Bank3 0: Disable Read Count to Bank3(Default)
B2RCENABLE Bank2 Read Count Enable	10	R/W	1: Enable Read Count to Bank2 0: Disable Read Count to Bank2(Default)
B1RCENABLE Bank1 Read Count Enable	9	R/W	1: Enable Read Count to Bank1 0: Disable Read Count to Bank1(Default)
B0RCENABLE Bank0 Read Count Enable	8	R/W	1: Enable Read Count to Bank0 0: Disable Read Count to Bank0(Default)
B7WCENABLE Bank7 WRite Count Enable	7	R/W	1: Enable Write Count to Bank7 0: Disable Write Count to Bank7(Default)
B6WCENABLE Bank6 WRite Count Enable	6	R/W	1: Enable Write Count to Bank6 0: Disable Write Count to Bank6(Default)
B5WCENABLE Bank5 WRite Count Enable	5	R/W	1: Enable Write Count to Bank5 0: Disable Write Count to Bank5(Default)
B4WCENABLE Bank4 WRite Count Enable	4	R/W	1: Enable Write Count to Bank4 0: Disable Write Count to Bank4(Default)
B3WCENABLE Bank3 WRite Count Enable	3	R/W	1: Enable Write Count to Bank3 0: Disable Write Count to Bank3(Default)
B2WCENABLE Bank2 WRite Count Enable	2	R/W	1: Enable Write Count to Bank2 0: Disable Write Count to Bank2(Default)

Table 5-11. DDR Metrics Counter Enable Register Bits (Cont'd)

Name	Offset	Access	Description
B1WCENABLE Bank1 WRite Count Enable	1	R/W	1: Enable Write Count to Bank1 0: Disable Write Count to Bank1(Default)
B0WCENABLE Bank0 WRite Count Enable	0	R/W	1: Enable Write Count to Bank0 0: Disable Write Count to Bank0(Default)

DDR Metrics Counter Clear (EBIU_DDRMCCL) Register

This 32-bit SMMR controls independent clearing of DDR metrics counters. Each bit in this register, when set to 1, clears the corresponding counter. Writing 0 in a bit position has no affect on the corresponding counter. This register is used to clear the corresponding counter(s) before starting them (see Table 5-12 and Table 5-13).

Table 5-12. DDR Metrics Counter Clear Register

Address	Register Name	Size	Reset Value
0xFFC0 0AC4	DDR Metrics Counter Clear Register	32	0x0000 0000

Table 5-13. DDR Metrics Counter Clear Register Bits

Name	Offset	Access	Description
Reserved	31:24	RO	Reads 0s
CG3COUNT Clear DDR Grant Count Register 3	23	R/W	1: Clear Grant Count Register 3 0: Do not Clear (Default)
CG2COUNT Clear DDR Grant Count Register 2	22	R/W	1: Clear Grant Count Register 2 0: Do not Clear (Default)
CG1COUNT Clear DDR Grant Count Register 1	21	R/W	1: Clear Grant Count Register 1 0: Do not Clear (Default)
CG0COUNT Clear DDR Grant Count Register 0	20	R/W	1: Clear Grant Count Register 0 0: Do not Clear (Default)
Reserved	19	RO	Reads 0

Table 5-13. DDR Metrics Counter Clear Register Bits (Cont'd)

Name	Offset	Access	Description
CARCOUNT Clear Total DDR Auto-Refresh Count	18	R/W	1: Clear Auto-Refresh Count 0: Do not Clear (Default)
CRWTACOUNT Clear Total DDR R/W Turn Around Count	17	R/W	1: Clear Turn Around Count 0: (Default)
CRACOUNT Clear Total DDR Row ACTIVATE Count	16	R/W	1: Clear Row Activate Count 0: Do not Clear (Default)
CB7RCOUNT Clear Bank7 Read Count	15	R/W	1: Clear Read Count to Bank7 0: Do not Clear (Default)
CB6RCOUNT Clear Bank6 Read Count	14	R/W	1: Clear Read Count to Bank6 0: Do not Clear (Default)
CB5RCOUNT Clear Bank5 Read Count	13	R/W	1: Clear Read Count to Bank5 0: Do not Clear (Default)
CB4RCOUNT Clear Bank4 Read Count	12	R/W	1: Clear Read Count to Bank4 0: Do not Clear (Default)
CB3RCOUNT Clear Bank3 Read Count	11	R/W	1: Clear Read Count to Bank3 0: Do not Clear (Default)
CB2RCOUNT Clear Bank2 Read Count	10	R/W	1: Clear Read Count to Bank2 0: Do not Clear (Default)
CB1RCOUNT Clear Bank1 Read Count	9	R/W	1: Clear Read Count to Bank1 0: Do not Clear (Default)
CB0RCOUNT Clear Bank0 Read Count	8	R/W	1: Clear Read Count to Bank0 0: Do not Clear (Default)
CB7WCOUNT Clear Bank7 Write Count	7	R/W	1: Clear Write Count to Bank7 0: Do not Clear (Default)
CB6WCOUNT Clear Bank6 Write Count	6	R/W	1: Clear Write Count to Bank6 0: Do not Clear (Default)
CB5WCOUNT Clear Bank5 Write Count	5	R/W	1: Clear Write Count to Bank5 0: Do not Clear (Default)
CB4WCOUNT Clear Bank4 Write Count	4	R/W	1: Clear Write Count to Bank4 0: Do not Clear (Default)
CB3WCOUNT Clear Bank3 Write Count	3	R/W	1: Clear Write Count to Bank3 0: Do not Clear (Default)

Table 5-13. DDR Metrics Counter Clear Register Bits (Cont'd)

Name	Offset	Access	Description
CB2WCOUNT Clear Bank2 Write Count	2	R/W	1: Clear Write Count to Bank2 0: (Default)
CB1WCOUNT Clear Bank1 Write Count	1	R/W	1: Clear Write Count to Bank1 0: Do not Clear (Default)
CB0WCOUNT Clear Bank0 Write Count	0	R/W	1: Clear Write Count to Bank0 0: Do not Clear (Default)

DDR READ Access Count (EBIU_DDRBRCx) Registers

Each of the following registers counts read accesses to the corresponding DDR SDRAM bank, when enabled. Bank4 through Bank7 imply banks in the second external memory bank.

- DDR Bank0 Read Count (`EBIU_DDRBRC0`) Register (Address: 0xFFC0 0A60)
- DDR Bank1 Read Count (`EBIU_DDRBRC1`) Register (Address: 0xFFC0 0A64)
- DDR Bank2 Read Count (`EBIU_DDRBRC2`) Register (Address: 0xFFC0 0A68)
- DDR Bank3 Read Count (`EBIU_DDRBRC3`) Register (Address: 0xFFC0 0A6C)
- DDR Bank4 Read Count (`EBIU_DDRBRC4`) Register (Address: 0xFFC0 0A70)
- DDR Bank5 Read Count (`EBIU_DDRBRC5`) Register (Address: 0xFFC0 0A74)

- DDR Bank6 Read Count (EBIU_DDRBRC6) Register (Address: 0xFFC0 0A78)
- DDR Bank7 Read Count (EBIU_DDRBRC7) Register (Address: 0xFFC0 0A7C)

DDR WRITE Access Count (EBIU_DDRBWCx) Registers

Each of the following registers counts write accesses to the corresponding DDR SDRAM bank, when enabled. Bank4 through Bank7 imply banks in the second external memory bank.

- DDR Bank0 Write Count Register (EBIU_DDRBWC0) (Address: 0xFFC0 0A80)
- DDR Bank1 Write Count Register (EBIU_DDRBWC1) (Address: 0xFFC0 0A84)
- DDR Bank2 Write Count Register (EBIU_DDRBWC2) (Address: 0xFFC0 0A88)
- DDR Bank3 Write Count Register (EBIU_DDRBWC3) (Address: 0xFFC0 0A8C)
- DDR Bank4 Write Count Register (EBIU_DDRBWC4) (Address: 0xFFC0 0A90)
- DDR Bank5 Write Count Register (EBIU_DDRBWC5) (Address: 0xFFC0 0A94)
- DDR Bank6 Write Count Register (EBIU_DDRBWC6) (Address: 0xFFC0 0A98)
- DDR Bank7 Write Count Register (EBIU_DDRBWC7) (Address: 0xFFC0 0A9C)

DDR Page ACTIVATE Count (EBIU_DDRACCT) Register

This register counts the total number of times the page activate command was issued to the DDR SDRAM, for all banks. (Address: 0xFFC0 0AA0)

DDR TURN AROUND Count (EBIU_DDRTACT) Register

This register counts the total number of times there was a turn around between read and write or between write and read commands, for all banks. (Address: 0xFFC0 0AA8)

DDR AUTO-REFRESH Count (EBIU_DDRARCT) Register

This register counts the total number of times the auto-refresh command was issued, for all banks. (Address: 0xFFC0 0AAC)

DDR Grant Count (EBIU_DDRGCx) Registers

There are four DDR grant count registers. These counters may be used to monitor how the four requesters (for example, EAB, DEB0, DEB1, DEB2) are granted access to the DDR memory.

- DDR Grant Count Register 0 (`EBIU_DDRGC0`)
  (Address: 0xFFC0 0AB0) If the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0 – this register counts the total number of times the EAB was granted access to DDR SDRAM.
- DDR Grant Count Register 1 (`EBIU_DDRGC1`)
  (Address: 0xFFC0 0AB4) If the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0 – this register counts the total number of times the DEB0 was granted access to DDR SDRAM.

- DDR Grant Count Register 2 (EBIU_DDRGC2)
  (Address: 0xFFC0 0AB8) If the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0 – this register counts the total number of times the DEB1 was granted access to DDR SDRAM.
- DDR Grant Count Register 3 (EBIU_DDRGC3)
  (Address: 0xFFC0 0ABC) If the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0 – this register counts the total number of times the DEB2 was granted access to DDR SDRAM.

More Grant Counter Options

The grant count registers can be configured to record grants in different ways, depending upon the DDR grant control field of the DDR metrics counter enable register (bit[25:24]). Table 5-14 enumerates different ways the programmer can configure these counters.

Table 5-14. DDR Grant Control Scheme

Grant Control Field[1:0]	Grant Count Register 1	Grant Count Register 2	Grant Count Register 3	Grant Count Register 4
00	Total EAB Grants	Total DEB0 Grants	Total DEB1 Grants	Total DEB2 Grants
01	Total EAB Grants	DEB0 WR Grants	DEB0 RD Grants	DEB0 Prefetch Grants
10	Total EAB Grants	DEB1 WR Grants	DEB1 RD Grants	DEB1 Prefetch Grants
11	Total EAB Grants	DEB2 WR Grants	DEB2 RD Grants	DEB2 Prefetch Grants

DDR Grant Count Control

The DDR grant count control field (bits[25:24]) in the EBIU_DDRMCEN register helps monitor arbitration activities inside the EBIU arbiter.

- When this field is set to 00, the DDR grant count registers 0, 1, 2, and 3 count the number of grants given to EAB, DEB0, DEB1, and DEB2 buses respectively for access requests to DDR SDRAM.
- When this field is set to 01, DDR grant count registers 1, 2, and 3 count the total number of grants given to DEB0, the number of grants given to DEB0 write requests, the number of grants given to DEB0 read requests and the number of grants given to DEB0 prefetch read requests, respectively. Grant count register 0 counts the number of grants given to EAB.
- When this field is set to 10, DDR grant count registers 1, 2, and 3 count the total number of grants given to DEB1, the number of grants given to DEB1 write requests, the number of grants given to DEB0 read requests and the number of grants given to DEB1 prefetch read requests, respectively. Grant count register 0 counts the number of grants given to EAB.
- When this field is set to 11, DDR grant count registers 1, 2, and 3 count the total number of grants given to DEB2, the number of grants given to DEB2 write requests, the number of grants given to DEB2 read requests and the number of grants given to DEB2 prefetch read requests, respectively. Grant count register 0 counts the number of grants given to EAB.

Asynchronous Memory Interface

The EBIU interface allows a view into a variety of memory and peripheral devices, including SRAM, ROM, EPROM, NOR flash memory, and FPGA/ASIC devices. Four asynchronous memory regions (banks) are supported. Each has a unique memory select associated with it, as shown in Table 5-15.

Table 5-15. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
AMS[3]	2C00 0000	2FFF FFFF
AMS[2]	2800 0000	2BFF FFFF
AMS[1]	2400 0000	27FF FFFF
AMS[0]	2000 0000	23FF FFFF

Although it is called asynchronous memory interface, each bank in the asynchronous memory region supports synchronous memory devices such as NOR flash memory. Each bank may be individually configured for one of three operating modes:

- Asynchronous read/write
- Asynchronous page mode read
- Synchronous burst read

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 64M bytes. Many code and data structures may fit within the confines of a single memory bank and not all of an enabled memory bank needs be populated.

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

The asynchronous memory signals are defined in Table 5-16. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. Certain pins switch between asynchronous and flash functions depending on the access mode selected for the memory bank being accessed. For example interfaces, see Chapter 19, “System Design”.

Table 5-16. Asynchronous Memory Interface Pins

Pin Name	Type	Asynchronous Function	FLASH Function	Changes with Mode?
ADDR25	O	Address Bus	Clock Output (CLK)	Yes
ADDR24–1	O	Address Bus	Address Bus	No
DATA15–0	I/O	Data Bus	Data Bus	No
$\overline{\text{AMSx}}$	O	Memory Select	Chip Enable (CE#)	No
$\overline{\text{ABE0}}$	O	Lower Byte Enable	--	Yes
$\overline{\text{ABE1}}$	O	Upper Byte Enable	--	Yes
$\overline{\text{AOE}}$	O	Output Enable	Address Valid (ADV#)	Yes
$\overline{\text{ARE}}$	O	Read Enable	Output Enable (OE#)	No
$\overline{\text{AWE}}$	O	Write Enable	Write Enable (WE#)	No
ARDY	O	Ready	Wait (WAIT#)	No

Asynchronous Memory Arbitration

The asynchronous memory arbiter accepts requests from the address resolution block for each of the DEB0, DEB1, DEB2, and the external access bus. The arbiter in the ASYNC follows an arbitration scheme similar to DDR, but simplified. The DMA reads and writes have the same priority, which eliminates the need for “forced write”. Also, there is no prefetch access in the asynchronous memory interface.

Table 5-17 summarizes the arbitration scheme for the asynchronous memory interface. The DEB_ARB_PRIORITY bits of the EBIU_DDRQUE register control the arbitration.

Table 5-17. Asynchronous Memory Interface Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Core TESTSET	Core TESTSET	Core TESTSET
Urgent DMA DEB0READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE
Normal DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 (USB) READ/WRITE	Normal DMA DEB2 (USB) READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE

The priority schemes described above are from the arbiter perspective. The priority schemes are followed by the arbiter only when they are ready to arbitrate, not when EBIU receives requests on the DEB or EAB buses. For example, a DEB bus may indicate Urgent during a request, but if the urgent signal goes away before the arbiter arbitrates, the DEB request is treated as regular request. Burst requests, from core, are arbitrated only in the first beat of a burst; for example, once processor core access is granted, it is granted for the whole burst.

Asynchronous Memory Interface Control Registers

The EBIU contains memory-mapped registers that control the access characteristics for each asynchronous memory bank. In addition, a status register is provided to reflect the arbiter status.

Table 5-18. EBIU Memory-Mapped Registers

Address	Register Name	Description
0xFFC0 0A00	EBIU_AMGCTL	"Asynchronous Memory Global Control Register (EBIU_AMGCTL)" on page 5-57
0xFFC0 0A04	EBIU_AMBCTL0	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 5-59
0xFFC0 0A08	EBIU_AMBCTL1	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 5-59
0xFFC0 0A0C	EBIU_AMBSCTL	"Memory Bank Select Control Register (EBIU_MBSCTL)" on page 5-63
0xFFC0 0A10	EBIU_ARBSTAT	"EBIU Arbitration Status Register (EBIU_ARBSTAT)" on page 5-69
0xFFC0 0A14	EBIU_MODE	"Memory Mode Control (EBIU_MODE) Register" on page 5-66
0xFFC0 0A18	EBIU_FCTL	"Flash Memory Bank Control (EBIU_FCTL) Register" on page 5-67
0xFFC0 0A1C	Reserved	Reserved

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

The `EBIU_AMGCTL` register configures global aspects of the controller. It contains bank enables and other information as described in this section. Do not program this register while the ASYNC is in use (for example, when code is being executed from this memory space).

The EBIU_AMGCTL register should be the last control register written-to when configuring the processor to access asynchronous memory-mapped devices.

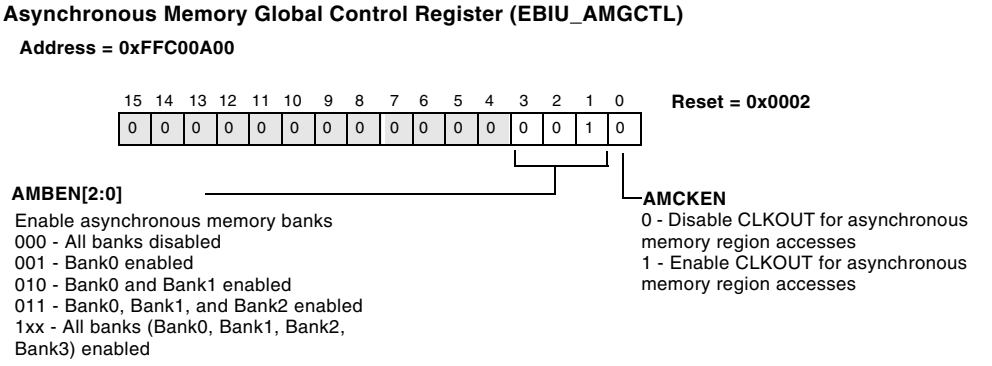


Figure 5-14. Asynchronous Memory Global Control Register

If a bus operation accesses a disabled asynchronous memory bank, the EBIU responds by acknowledging the transfer and asserting the error signal on the requesting bus. The error signal propagates back to the requesting bus master. This generates a hardware exception to the core, if it is the requester. For DMA-mastered requests, the error is captured in the respective status register.

If a bank is not fully populated with memory, then the memory likely aliases into multiple address regions within the bank. This aliasing condition is not detected by the EBIU, and no error response is asserted.

For external devices that need a clock, CLKOUT can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)

The EBIU asynchronous memory controller has two memory bank control registers (EBIU_AMBCTL0 and EBIU_AMBCTL1). They contain bits for counters for setup, strobe, and hold time, bits to determine memory type and size, and bits to configure use of ARDY. The configuration in these registers applies in all three operating modes. These registers should not be programmed while the ASYNC is in use.

The timing characteristics of the asynchronous memory interface can be programmed using these four parameters.

- Setup – The time between the beginning of a memory cycle ($\overline{\text{AMSx}}$ 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 ($\overline{\text{AMSx}}$ high).

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

Setup ≥ 1 cycle

Read Access ≥ 1 cycle

Write Access ≥ 1 cycle

Hold ≥ 0 cycle

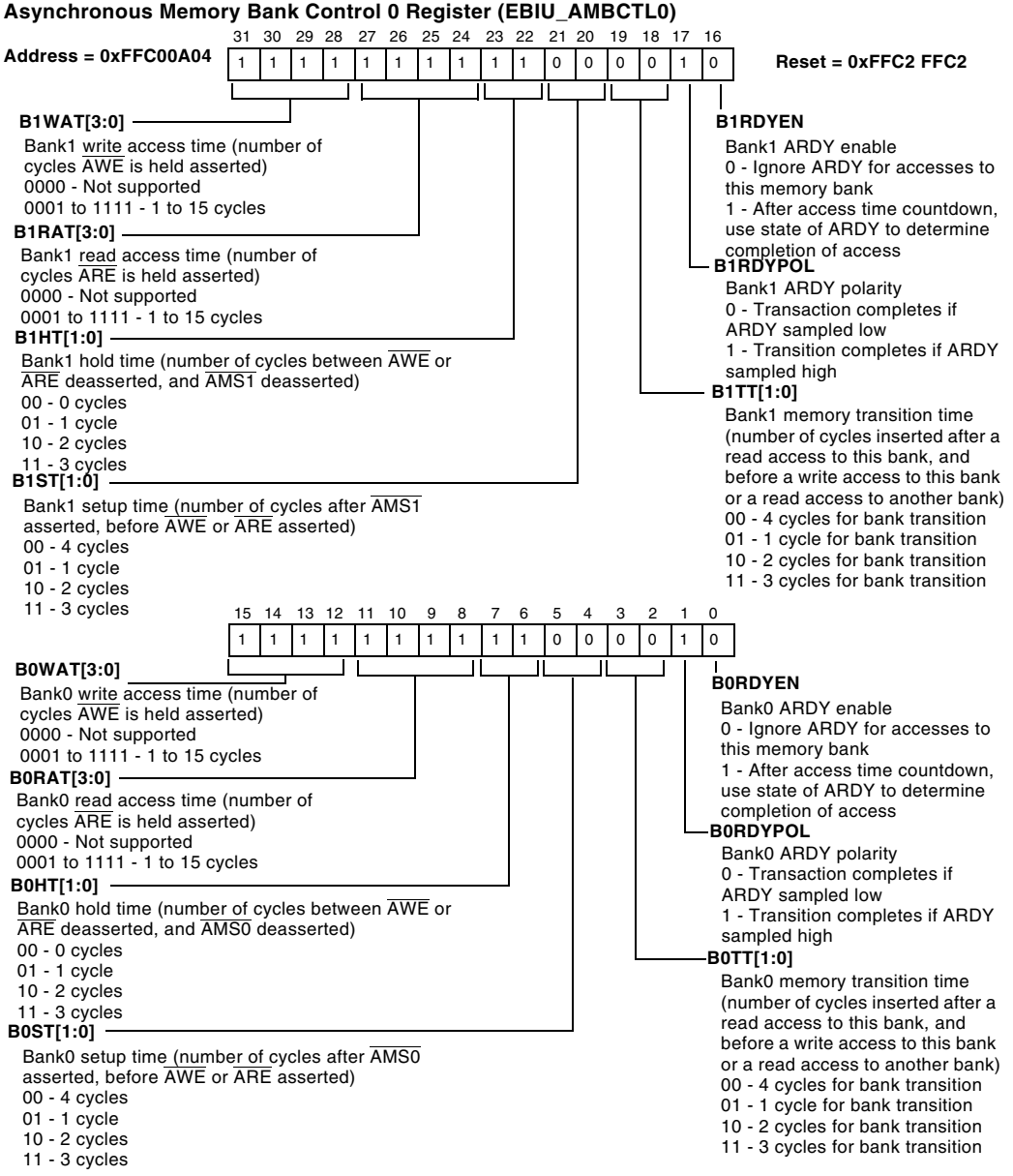


Figure 5-15. Asynchronous Memory Bank Control 0 Register

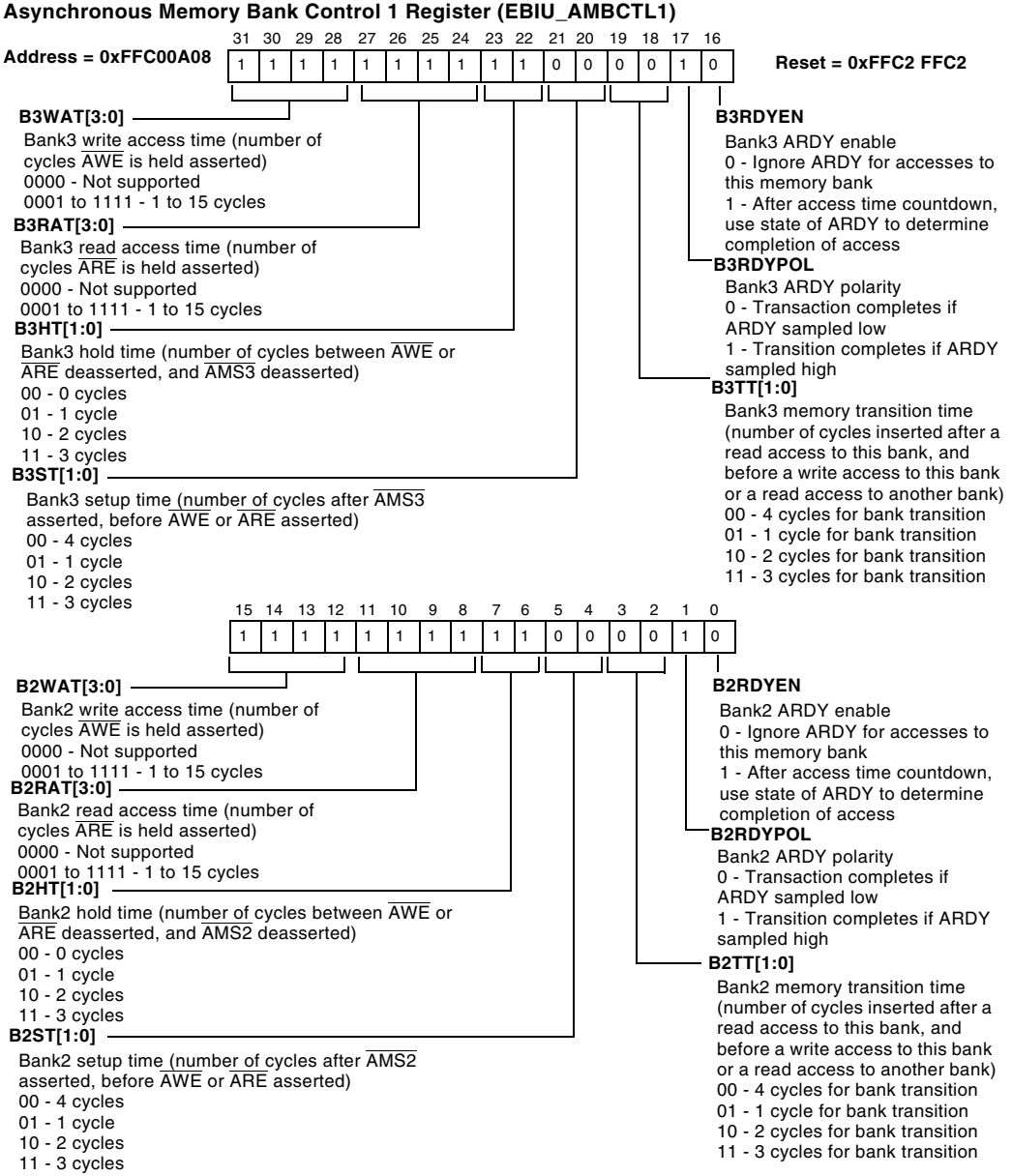


Figure 5-16. Asynchronous Memory Bank Control 1 Register

Avoiding Bus Contention

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

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

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

ARDY Input Control

Each bank can be programmed to sample the ARDY input after the read or write access timer has counted down or to ignore this input signal. If enabled and disabled at the sample window, ARDY can be used to extend the access time as required.

The ARDY input is treated as an asynchronous input, however it must reach the desired value (either asserted or deasserted) more than one SCLK cycle before the scheduled rising edge of $\overline{\text{AWE}}$ or $\overline{\text{ARE}}$. This determines whether the access is extended or not. Once the transaction has been extended as a result of ARDY being sampled in the "busy" state, the transaction will then complete in the cycle after ARDY is subsequently sampled in the "ready" state.

The polarity of ARDY is programmable on a per-bank basis. Since ARDY is not sampled until an access is in progress to a bank in which the ARDY enable is asserted, ARDY does not need to be driven by default. For more information, see "Adding Additional Wait States" on page 5-77.

When using flash memory, the $\overline{\text{WAIT}}$ input should be connected to ARDY.

Memory Bank Select Control Register (EBIU_MBSCTL)

External FIFO devices often do not have a separate chip select pin. As a result, for a read, the FIFO's output enable ($\overline{\text{OE}}$) pin must be connected the OR (negative AND) of the $\overline{\text{AMS}}$ and the $\overline{\text{ARE}}$. Similarly, the write case requires an OR between $\overline{\text{AMS}}$ and $\overline{\text{AWE}}$. The Blackfin processor provides this function in the EBIU so that an external OR gate is not required. The appropriate $\overline{\text{AMS}}$ function can be selected for each memory bank region in the EBIU_MBSCTL register.

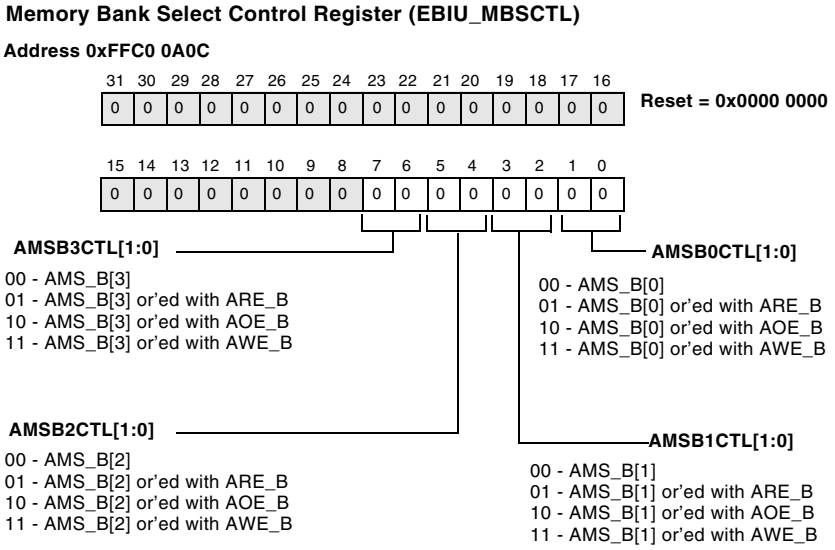


Figure 5-17. Memory Bank Select Control Register

Flash Memory Bank Control Registers (EBIU_FCTL, EBIU_MODE)

The asynchronous memory controller (ASYNC) also has two flash memory bank control registers.

- "Flash Memory Bank Control (EBIU_FCTL) Register" on page 5-67
- "Memory Mode Control (EBIU_MODE) Register" on page 5-66

They contain bits for mode selection, page access configuration, and synchronous access configuration. These registers should not be programmed while the ASYNC is in use.

Booting From Page Mode or Synchronous Flash

The EBIU resets to asynchronous mode access. This allows slow asynchronous access to any device during booting without configuration of the EBIU control registers. Synchronous burst mode and asynchronous page mode flash devices power up in asynchronous access mode and thus support an initial access of this type. Once configuration information is read from the external device, the boot code may select a higher performance operating mode.

Access Mode Selection

The EBIU may be configured for standard asynchronous mode access, asynchronous flash mode, asynchronous page mode access, or synchronous burst access. Asynchronous mode access should be used for most devices other than flash. Burst mode and page mode should only be used for read accesses. Flash mode (MODE = 01) must be used for all writes to flash devices. The burst mode and page mode controls have no effect unless the corresponding access mode is selected.

Pin functionality and supported device width change with mode, as described in Table 5-19.

Table 5-19. EBIU Pin Configuration by Mode

Mode	$\overline{\text{AOE}}$	ADDR[25]	Device Width
Asynchronous	$\overline{\text{AOE}}$	ADDR[25]	8 or 16 bit
Asynchronous flash	$\overline{\text{ADV}}$	ADDR[25]	16 bit
Asynchronous page	$\overline{\text{ADV}}$	ADDR[25]	16 bit
Synchronous burst	$\overline{\text{ADV}}$	CLK	16 bit

Memory Mode Control (EBIU_MODE) Register

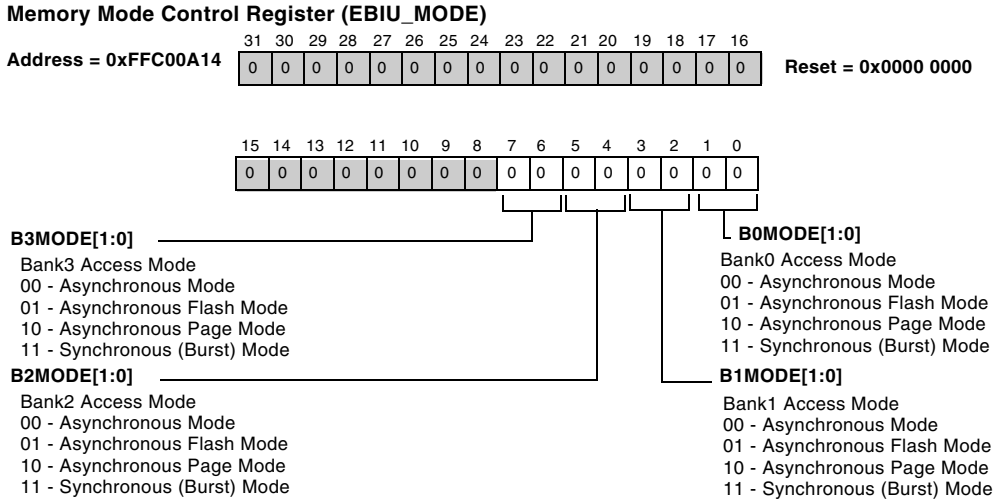


Figure 5-18. Memory Mode Control Register

Asynchronous Flash Mode

When the access selected mode is asynchronous flash (MODE = 01), external bank accesses operate exactly the same as in standard asynchronous mode, except for the pin configuration. This mode should be used when accessing burst devices in non-read array modes.

Flash Memory Bank Control (EBIU_FCTL) Register

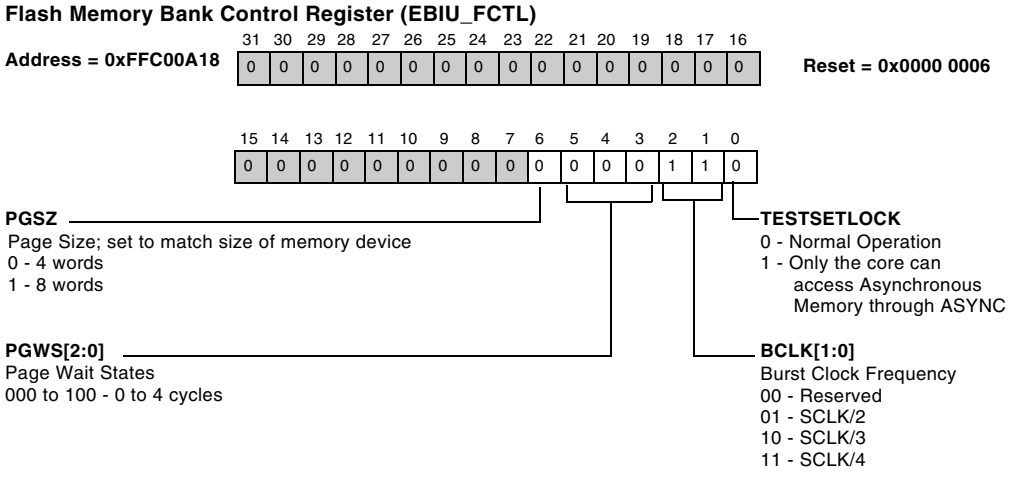


Figure 5-19. Flash Memory Bank Control Register

Asynchronous Page Mode

When asynchronous page mode access is selected (MODE = 10), asynchronous page reads are enabled. Page sizes of 4 or 8 words are supported. When performing a page mode read, the first access in the page proceeds according to the read access time configured in EBIU_AMBCTLx. This opens the page. Subsequent reads in that page extend the strobe time by one SCLK plus the number of page wait states. Page mode access is only supported for back-to-back accesses, such as cache line fills (16 words), 64-bit instruction reads (4 words) and 32-bit DMA reads (2 words).

Synchronous Burst Mode

When synchronous mode access is selected (MODE = 11), synchronous reads are enabled. The burst clock frequency can be configured for SCLK/2, SCLK/3 or SCLK/4.

This is the frequency of the clock output and determines the frequency of latching data for subsequent beats of a burst. It does not affect any of the other timing parameters (which are still determined by EBIU_AMBCTLx).

During the setup time of an access, $\overline{\text{ADV}}$ is asserted and the burst clock begins running. The flash device must be configured to latch the address on the rising edge of the clock. $\overline{\text{ADV}}$ is asserted for the entire setup time. The first rising edge of CLK occurs one SCLK cycle before setup ends.

The setup time must be configured appropriately with respect to the SCLK to CLK (burst clock) ratio, as follows.

- if SCLK to CLK ratio is 4:1 then setup time = 3 SCLK cycles
- if SCLK to CLK ratio is 3:1 then setup time = 2 SCLK cycles
- A minimum of 2 SCLK cycles must be programmed regardless of SCLK to CLK ratio.

Once the address is latched, the initial burst access occurs based on the read access timing for that bank. The strobe time is then extended by a burst clock duration for each subsequent beat of the burst. Any access in the burst may be extended by connecting the flash $\overline{\text{WAIT}}$ to ARDY. The flash device must be configured to deassert ARDY at the same time that data is valid. Depending on the flash behavior, it may be necessary to disable the ARDY input before asynchronous read or write accesses.

The synchronous read may be burst or single mode, depending on the type of transfer requested. Burst access is only supported for back-to-back reads, such as cache line fills (16 words), 64-bit instruction reads (4 words), and 32-bit DMA reads (2 words). Burst access is not supported for 8-bit accesses. To support any of these burst types, the flash device must be configured for 16-word wrapping burst mode.

When programming the ASYNC, before setting the ASYNC to synchronous burst mode (MODE = 11), it is necessary to do SSYNC instruction and then wait for (BxST + BxWAT + BxHT) SCLK cycles, where x is the bank being accessed and the terms are the configuration values from EBIU_AMBCTL0 or EBIU_AMBCTL1. This is to prevent the potential contention of previous FLASH device operation and the upcoming mode change.

EBIU Arbitration Status Register (EBIU_ARBSTAT)

When the external flash device is put in non-read array mode for programming, erasing, or checking status, accesses to memory locations in the flash do not return the stored data. As a result, an arbitration locking mechanism is provided to allow the core to prevent DMA access during these operations.

Specifically, the EBIU may be configured to only allow DSP core accesses to the asynchronous memory banks, by setting the TESTSETLOCK bit in EBIU_FCTL. Once this bit is set, only the core can win arbitration for future accesses. Depending on the speed of any outstanding accesses, it may take many cycles before the arbitration lock takes effect. The EBIU_ARBSTAT register contains a status bit to indicate when the arbiter is locked. Once the arbiter is locked, any DMA access to the asynchronous memory banks is stalled until the TESTSETLOCK bit is cleared.

It is recommended that software manage flash memory programming and DMA activities to prevent stalling of the DMA with arbiter locked status.

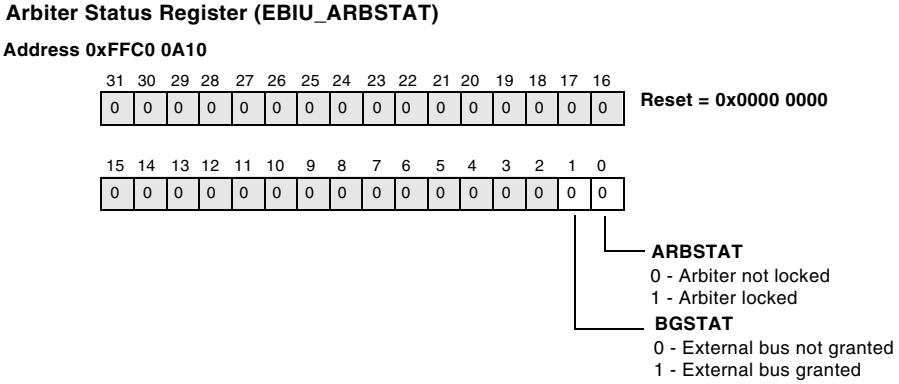


Figure 5-20. Arbiter Status Register

Programmable Timing Characteristics

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

Asynchronous Accesses by Core Instructions

Some asynchronous memory accesses are caused by core instructions of the type:

```
R0.L = W[P0++] ; /* Read from Asynchronous Memory, where P0
points to a location in Asynchronous Memory */
```

or:

```
W[P0++] = R0.L ; /* Write to Asynchronous Memory */
```

Asynchronous Reads

Figure 5-21 shows two core-initiated asynchronous read bus cycles to the same bank, with timing programmed with setup = 1 cycle, read access = 3 cycles, hold = 2 cycles, and transition time = 1 cycle.

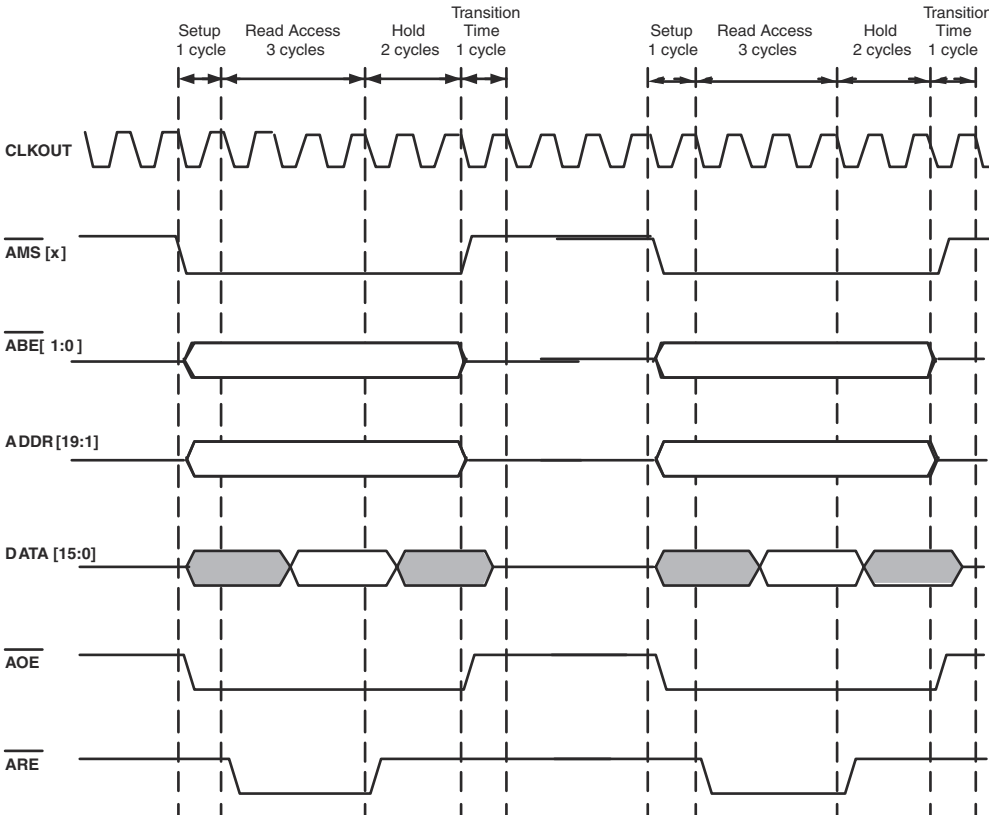


Figure 5-21. Core-Initiated Asynchronous Read Bus Cycles

Asynchronous read bus cycles proceed as:

- At the start of the setup period, $\overline{\texttt{AMSx}}$, the address bus, and $\overline{\texttt{ABE1-0}}$ become valid, and $\overline{\texttt{AOE}}$ asserts.
- At the beginning of the read access period and after the setup cycle, $\overline{\texttt{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the falling edge of CLKOUT. The $\overline{\texttt{ARE}}$ pin deasserts after the falling edge.
- At the end of the hold period, $\overline{\texttt{AOE}}$ and $\overline{\texttt{AMSx}}$ deassert.

Unless another read of the same memory bank is queued internally, the ASYNC appends the programmed number of memory transition time cycles.

Asynchronous Writes

Figure 5-22 shows two core-initiated asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

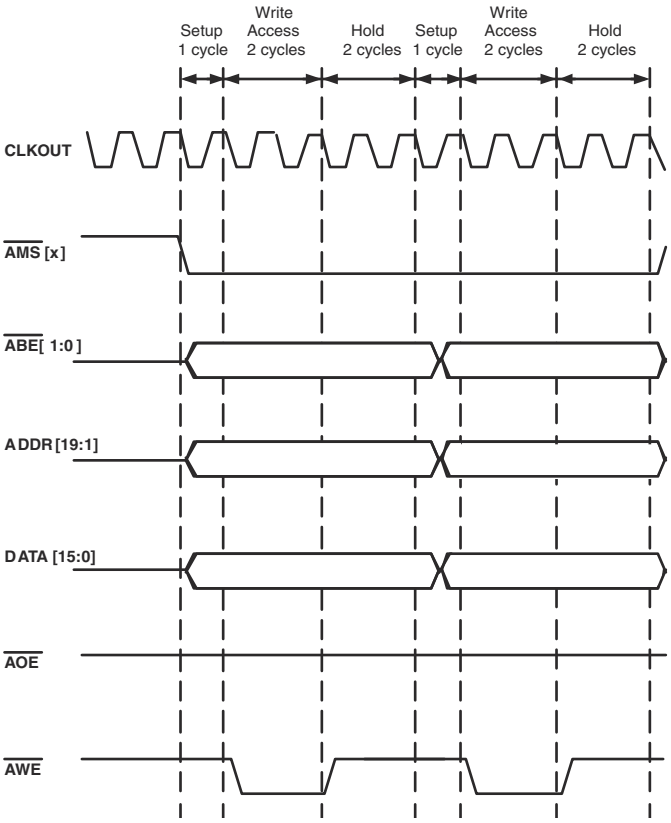


Figure 5-22. Core-Initiated Asynchronous Write Bus Cycles

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ remains low for the next setup period of the next access.

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$ is still asserted. The address and data buses and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

Figure 5-23 shows two higher-speed asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 0 cycles, and transition time = 1 cycle.

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

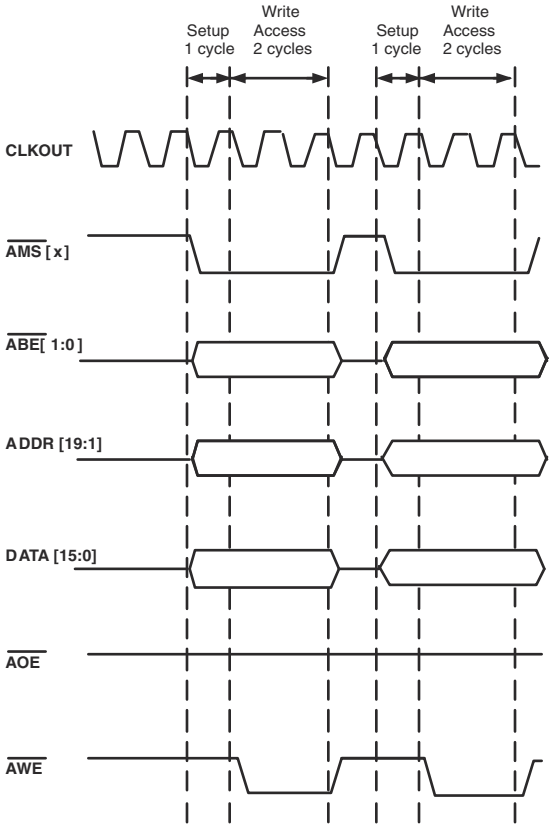


Figure 5-23. High Speed Core-Initiated Asynchronous Write Bus Cycles

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

Asynchronous Writes Followed by Reads

Figure 5-24 shows an asynchronous write bus cycle followed by two asynchronous read cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, read access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

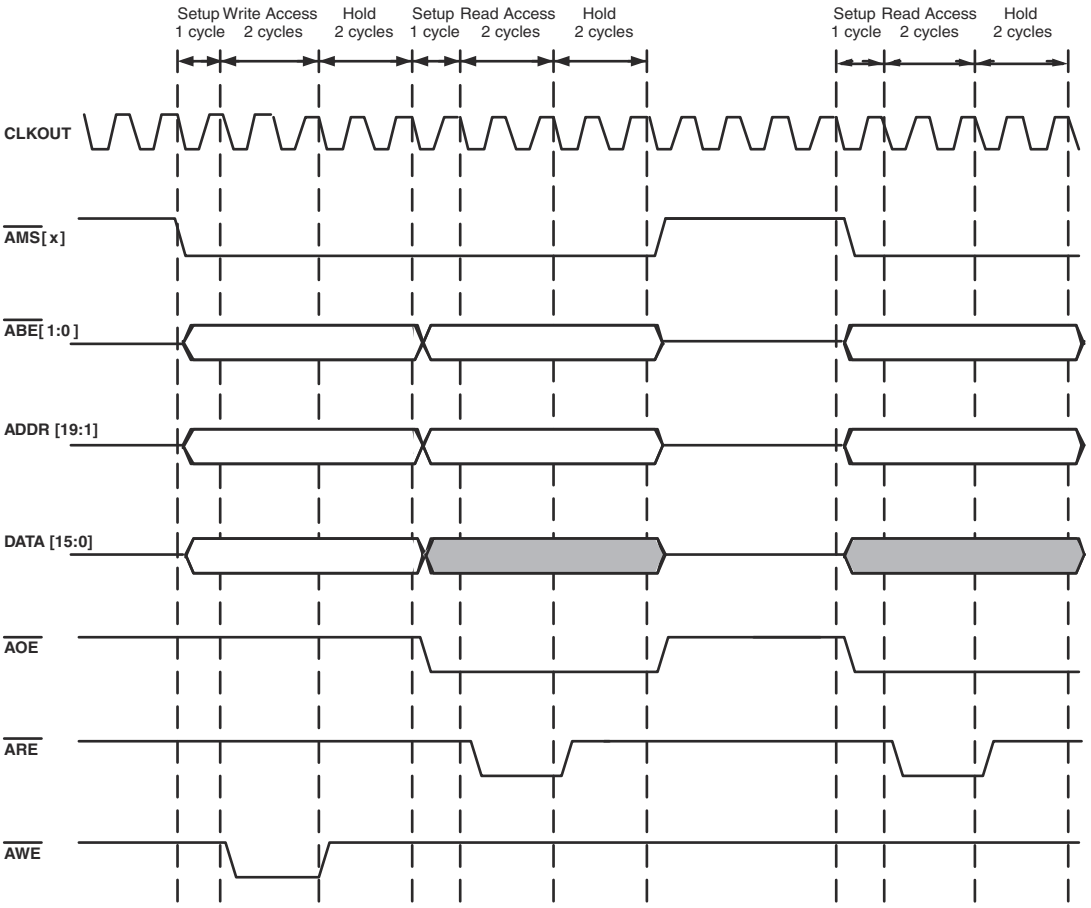


Figure 5-24. Core-Initiated Write and Read Bus Cycles

The asynchronous write bus cycles proceed as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts and $\overline{\text{AMSx}}$ remains low for the setup period of the next access.

The first asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$ is still asserted. The address bus, and $\overline{\text{ABE[1:0]}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period, $\overline{\text{ARE}}$ asserts.
- 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.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMSx}}$ deassert.

The second asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, and $\overline{\text{ABE1-0}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period, $\overline{\text{ARE}}$ asserts again.
- 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.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMSx}}$ deassert.

Unless another read of the same memory bank is queued internally, the ASYNC appends the programmed number of memory transition time cycles.

Adding Additional Wait States

The ARDY pin is used to insert extra wait states. An example of this behavior is shown in Figure 5-25 on page 5-77, 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.

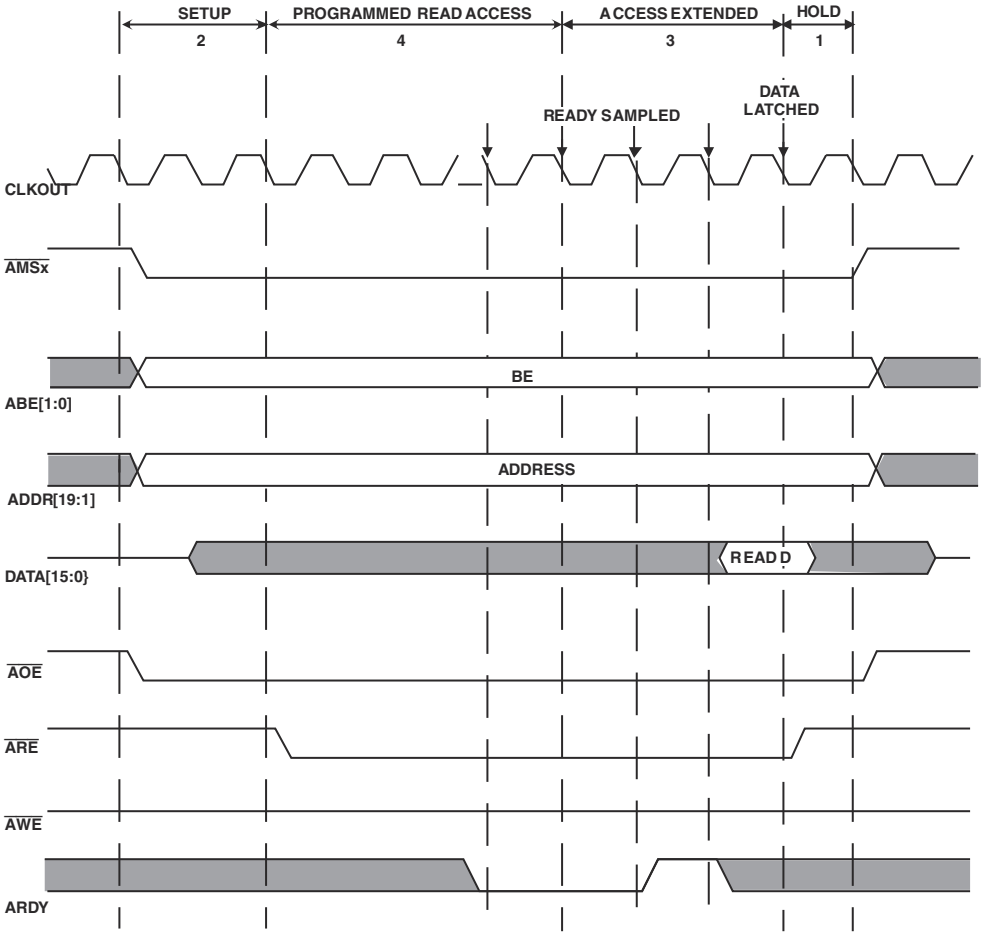


Figure 5-25. Inserting Wait States Using ARDY

Asynchronous Flash Mode Writes and Reads

Figure 5-26 shows an asynchronous flash write bus cycle followed by a read bus cycle to the same bank. Timing is programmed with setup = 1 cycle, write access = 2 cycles, read access = 2 cycles, hold =2 cycles, and transition = 1 cycle. The bus cycles are identical to the asynchronous mode case, except for the behavior of $\overline{AOE}$. In this case, $\overline{AOE}$ is used to indicate a valid address ($\overline{ADV}$).

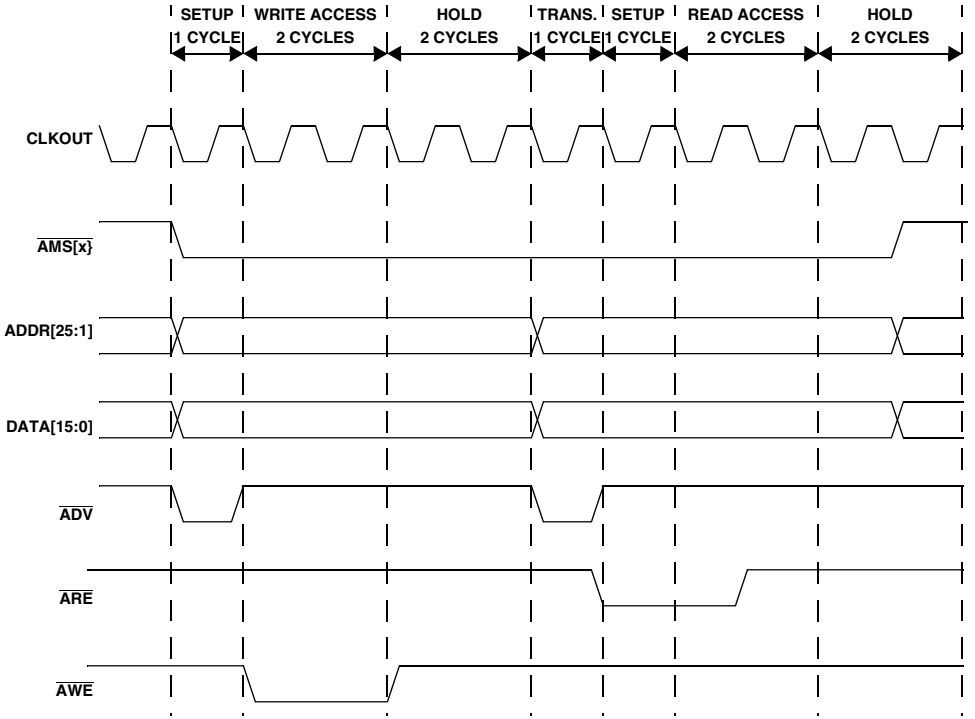


Figure 5-26. Asynchronous Flash Write and Read Bus Cycle

Asynchronous Page Mode Reads

Figure 5-27 shows an asynchronous page read bus cycle. Timing is programmed with setup = 1 cycle, read access = 3 cycles, hold =1 cycle, and transition = 1 cycle. One wait state (as specified in the PGWS field of the EBIU_FCTL register) is added to each access in the open page. AOE is used to indicate a valid address (ADV).

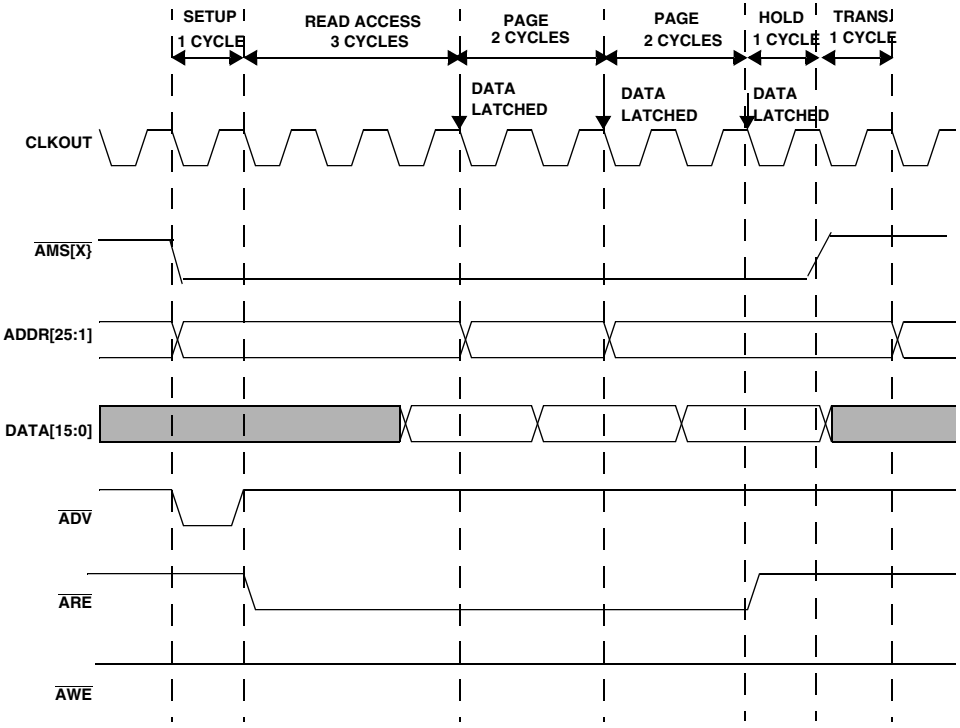


Figure 5-27. Asynchronous Page Mode Read Bus Cycle

Note: Asynchronous Page Mode is only valid for read operations.

Synchronous Burst Mode Read

Figure 5-28 shows a synchronous burst read bus cycle. Timing is programmed with setup = 3 cycles, read access = 2 cycles, hold = 1 cycle, and transition = 1 cycle. The burst clock frequency is SCLK/2. The initial burst access is extended using ARDY and the subsequent beats of the burst are latched on every rising CLK edge. $\overline{\text{AOE}}$ is used to indicate a valid address ($\overline{\text{ADV}}$) and ADDR25 (pin ADDR25) is used as the burst clock (CLK).

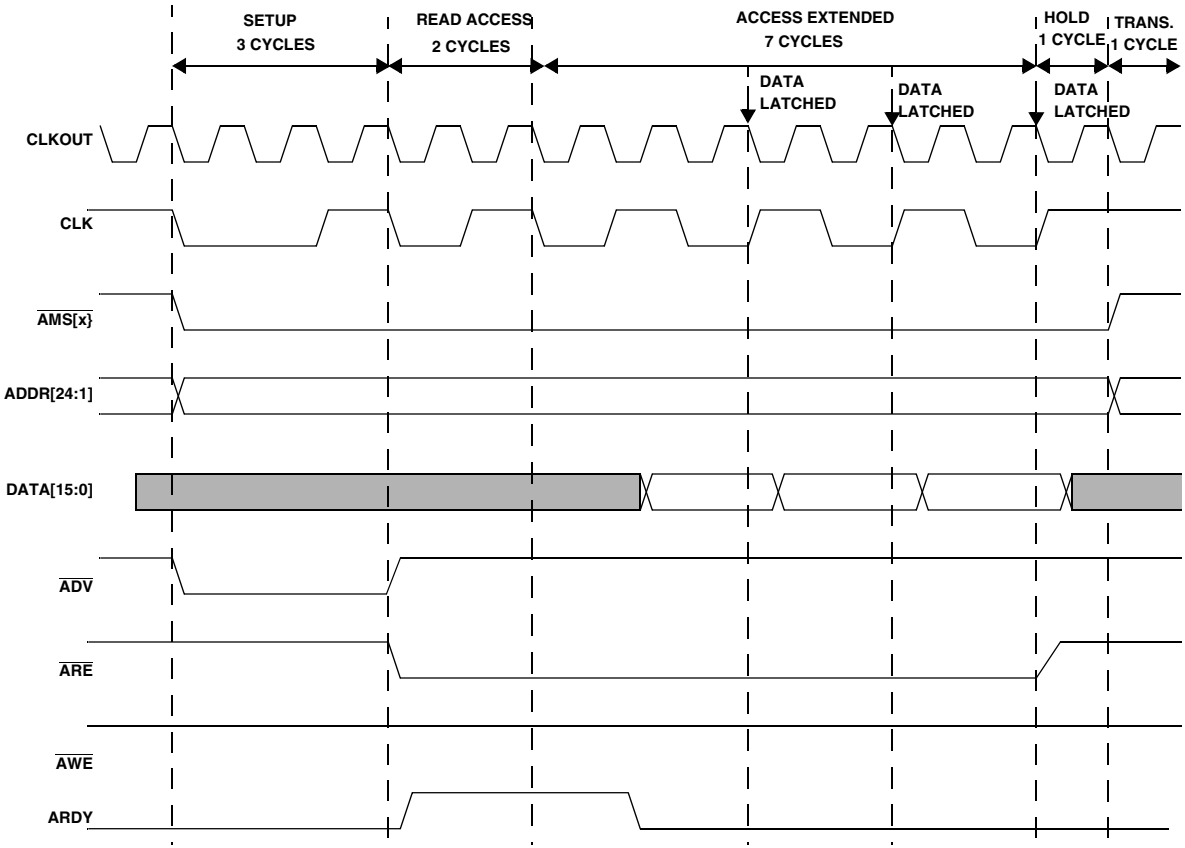


Figure 5-28. Synchronous Burst Mode Read Bus Cycle

Note: Synchronous mode is only valid for read operations, but does support both burst and non-burst operations.

Bus Request and Grant

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

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

- Three-stating the data and address buses and the asynchronous memory control signals. The synchronous memory control signals can optionally be three-stated.
- Asserting the bus grant ($\overline{\text{BG}}$) signal.

The processor may halt program execution if the bus is granted to an external device and an instruction fetch or data read/write request is made to external memory. When the external device releases $\overline{\text{BR}}$, the processor deasserts $\overline{\text{BG}}$ and continues execution from the point at which it stopped.

The processor asserts the $\overline{\text{BGH}}$ pin when it is ready to start another external port access, but is held off because the bus was previously granted. When the bus is granted, the `BGSTAT` bit in the `EBIU_ARBSTAT` register is set. This bit can be used by the processor to check the bus status to avoid initiating a transaction that would be delayed by the external bus grant.

6 SYSTEM INTERRUPTS

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

The chapter includes the following sections:

- "Overview" on page 6-1
- "Interfaces" on page 6-2
- "Description of Operation" on page 6-6
- "Programming Model" on page 6-22
- "System Interrupt Controller Registers" on page 6-24
- "Programming Examples" on page 6-40

Overview

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

Features

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

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

Interfaces

Figure 6-1, Figure 6-2, and Figure 6-3 provide an overview of how the individual peripheral interrupt request lines connect to the SIC. They also show how the 12 interrupt assignment registers (SIC_IARx) control the assignment to the 9 available peripheral request inputs of the CEC.

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

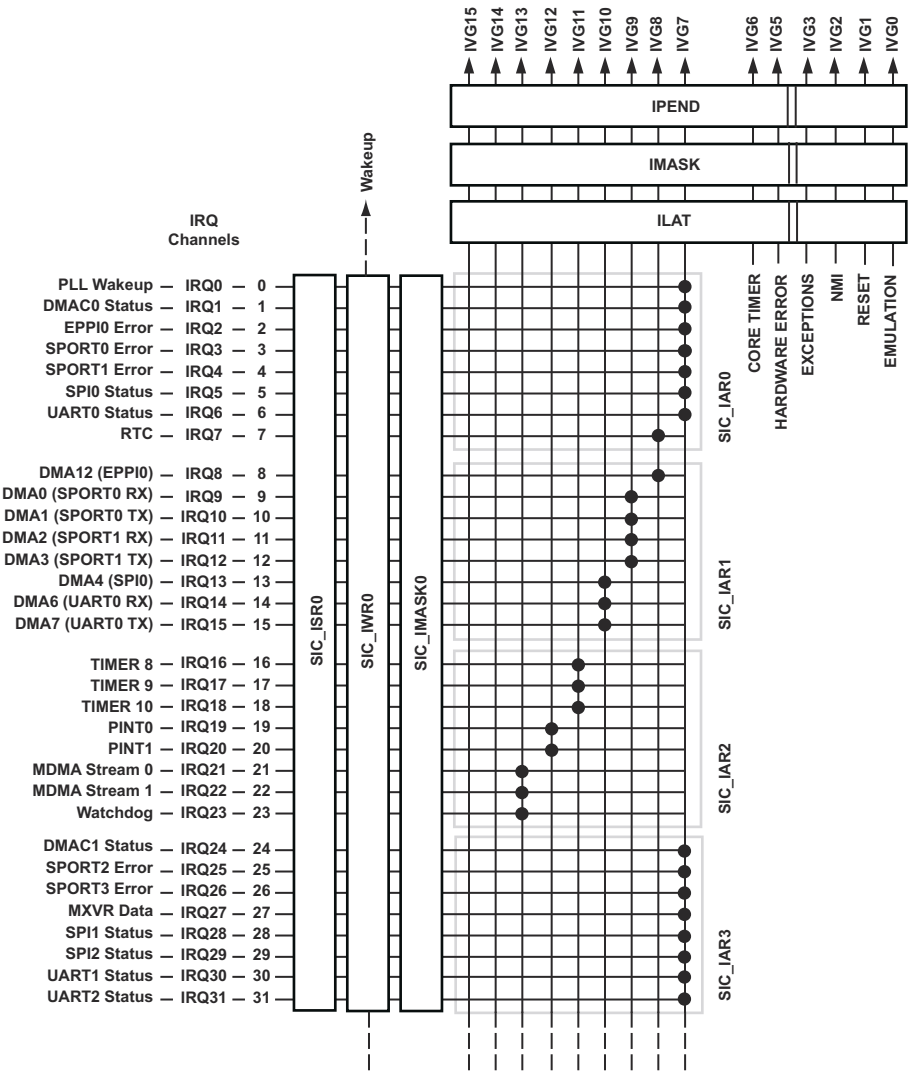


Figure 6-1. Interrupt Routing Overview Part 1 of 3

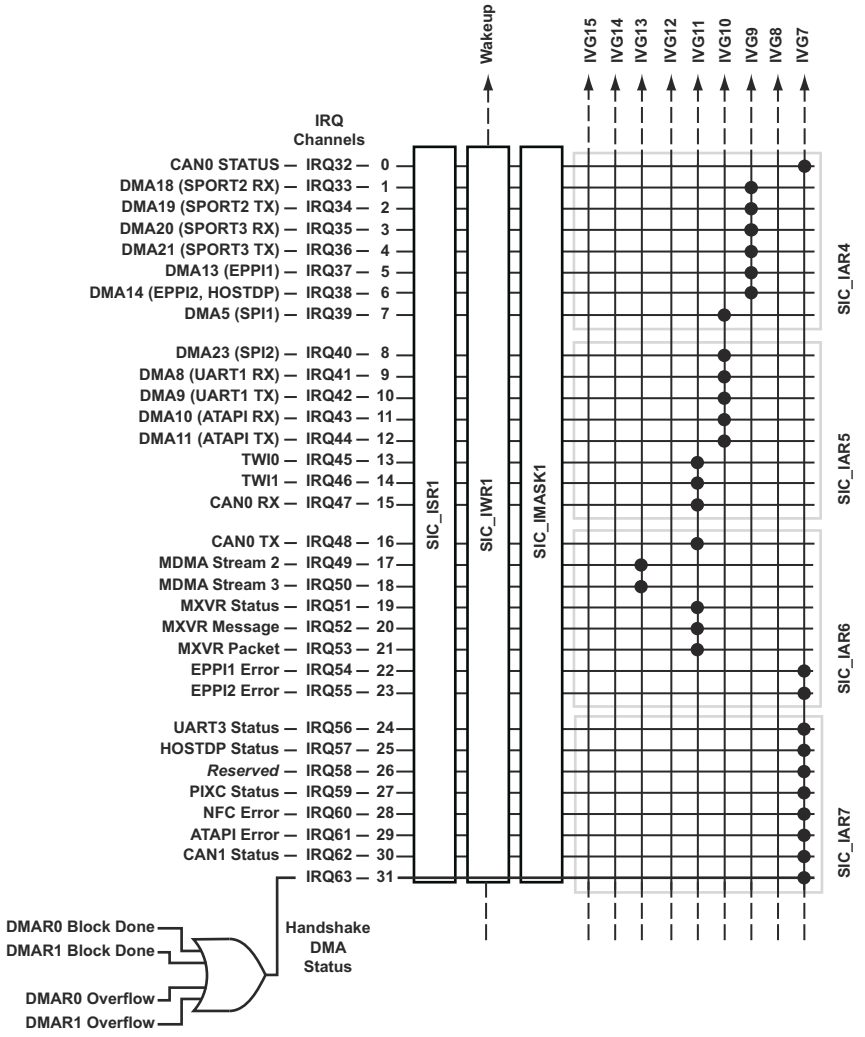


Figure 6-2. Interrupt Routing Overview Part 2 of 3

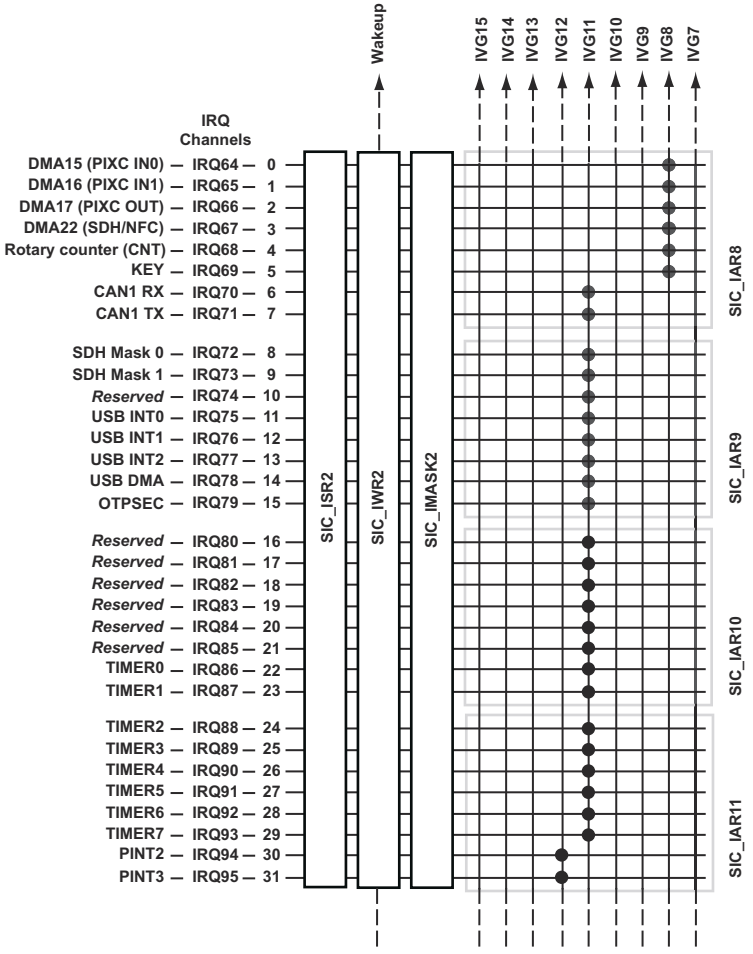


Figure 6-3. Interrupt Routing Overview Part 3 of 3

Description of Operation

The following sections describe the operation of the system interrupts.

Events and Sequencing

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

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

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

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

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

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be preempted 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 6-1.

Table 6-1. System and Core Event Mapping

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

Table 6-1. System and Core Event Mapping (Cont'd)

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

Table 6-1. System and Core Event Mapping (Cont'd)

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

Table 6-1. System and Core Event Mapping (Cont'd)

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

It is common for applications to reserve the lowest or the two lowest priority interrupts (`IVG14` and `IVG15`) for software interrupts, leaving eight or seven prioritized interrupt inputs (`IVG7` – `IVG13`) for peripheral purposes. Refer to Table 6-1.

The system interrupt to core event mappings shown in Table 6-1 are the default values at reset and can be changed by software.

System Peripheral Interrupts

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

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

The nine general-purpose interrupt inputs (IVG7–IVG15) of the core event controller have fixed priority. The IVG0 channel has the highest priority and IVG15 has the lowest priority. Therefore, the interrupt assignment in the SIC_IARx registers not only groups peripheral interrupts, but it also programs their priority by assigning them to individual IVG channels. However, the relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the system interrupt assignment register (SIC_IARx) settings, as detailed in Figure 6-7 on page 6-26 through Figure 6-18 on page 6-31. If more than one interrupt source is mapped to the same interrupt, they are logically OR'ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

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

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

The system interrupt mask register (SIC_IMASKx, shown in Figure 6-19 on page 6-32 through Figure 6-21 on page 6-34) allows software to mask any peripheral interrupt source at the SIC level. This functionality is independent of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of SIC_IMASKx are all 0s to mask off all peripheral interrupts. Turning off a system interrupt mask and enabling the particular interrupt is performed by writing a 1 to a bit location in the SIC_IMASKx register.

The SIC includes a read-only system interrupt status register (SIC_ISRx) with individual bits which correspond to one of the peripheral interrupt sources. See Figure 6-24 on page 6-37. When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input is deasserted, the respective bit in the system interrupt status register is cleared. Note for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt is deasserted.

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

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

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

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

Peripheral DMA channels are mapped in a fixed manner to the peripheral interrupt IDs. However, the assignment between peripherals and DMA channels is freely programmable with the DMAx_PERIPHERAL_MAP registers. Table 6-2 on page 6-16, Figure 6-4 on page 6-14, and Figure 6-5 on page 6-15 show the default DMA assignment. For more information on DMA, see Chapter 7, "Direct Memory Access". Once a peripheral is assigned to a DMA channel it uses the new DMA channel's interrupt ID regardless of whether DMA is enabled or not. Therefore, clean DMAx_PERIPHERAL_MAP management is required even if the DMA is not used. The default setup should be the best choice for all non-DMA applications.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state or from sleep mode to optionally process the interrupt, simply by enabling the appropriate bit in the system interrupt wakeup-enable register (SIC_IWRx, refer to Figure 6-25 on page 6-38). If a peripheral interrupt source is enabled in the SIC_IWRx register and the core is idled or in sleep mode, the interrupt causes the DPMC to initiate the core wakeup sequence in order to optionally process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see Chapter 18, "Dynamic Power Management".

The SIC_IWRx register has no effect unless the core is idled or in sleep mode. By default, all interrupts generate a wakeup request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as for a SPORTx transmit interrupt. The SIC_IWRx 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 the SIC_IWRx register but masked-off in the SIC_IMASKx register, the core wakes up if it is idled or in sleep mode, but it does not generate an interrupt.

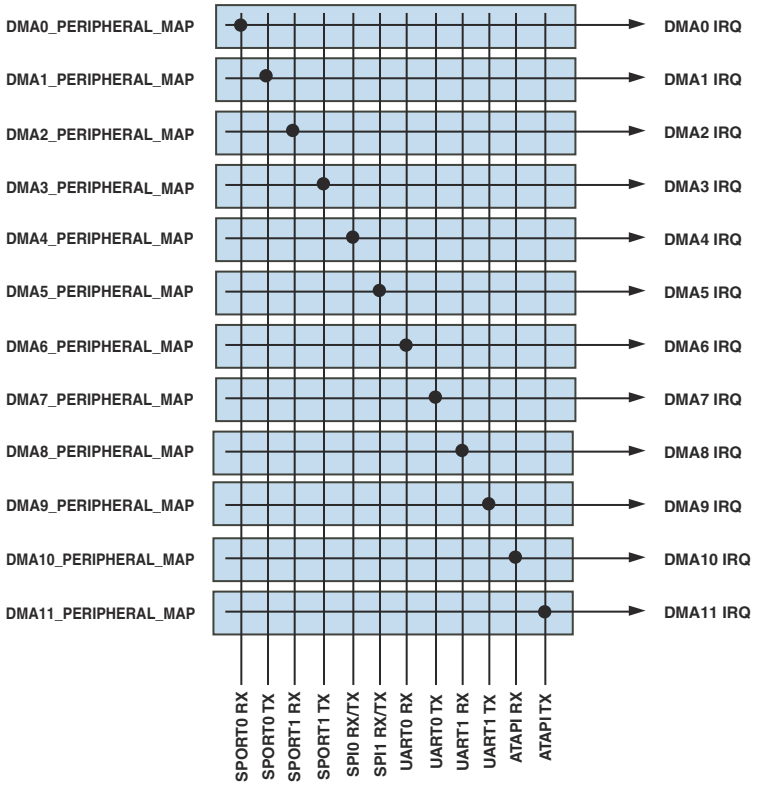


Figure 6-4. Default Peripheral-to-DMA Mapping (DMAC0 Controller)

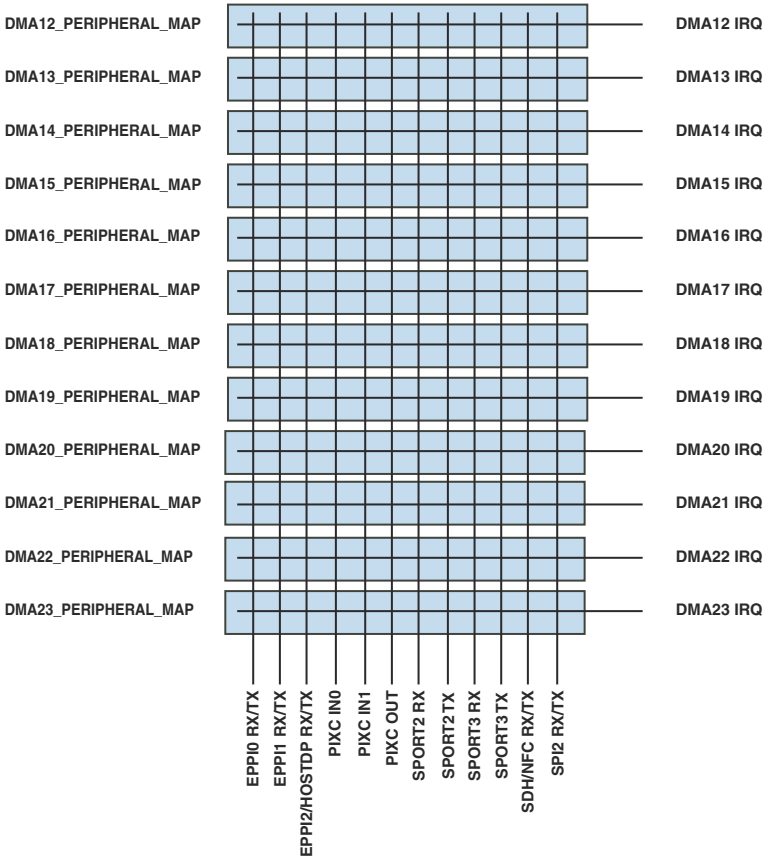


Figure 6-5. Default Peripheral-to-DMA Mapping (DMAC1 Controller)

Table 6-2 shows the peripheral interrupt events, the default mapping of each event, the peripheral interrupt ID used in the system interrupt assignment registers (`SIC_IARx`), and the core interrupt ID. See "System Interrupt Assignment (SIC_IARx) Registers" on page 6-25.

Table 6-2. System Interrupt Controller (SIC)

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

Table 6-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
Timer 8 Interrupt	16	IVG11	4	SIC_IAR2	SIC_IWR0, SIC_ISR0 & SIC_IMASK0
Timer 9 Interrupt	17	IVG11	4		
Timer 10 Interrupt	18	IVG11	4		
Pin Interrupt 0 (PINT0)	19	IVG12	5		
Pin Interrupt 1 (PINT1)	20	IVG12	5		
MDMA Stream 0 Interrupt	21	IVG13	6		
MDMA Stream 1 Interrupt	22	IVG13	6		
Software Watchdog Timer Interrupt	23	IVG13	6		
DMAC1 Status (generic)	24	IVG7	0	SIC_IAR3	
SPORT2 Error Interrupt	25	IVG7	0		
SPORT3 Error Interrupt	26	IVG7	0		
MXVR Synchronous Data Interrupt	27	IVG7	0		
SPI1 Status Interrupt	28	IVG7	0		
SPI2 Status Interrupt	29	IVG7	0		
UART1 Status Interrupt	30	IVG7	0		
UART2 Status Interrupt	31	IVG7	0		

Table 6-2. System Interrupt Controller (SIC) (Cont'd)

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

Table 6-2. System Interrupt Controller (SIC) (Cont'd)

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

Table 6-2. System Interrupt Controller (SIC) (Cont'd)

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

Table 6-2. System Interrupt Controller (SIC) (Cont'd)

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

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 6-2.

UART2 and UART3 are not assigned to peripheral channels by default. To assign one of these peripherals to a DMA channel, refer to Table 7-1 on page 7-10.

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 6-2 on page 6-16 are acceptable, then interrupt initialization involves only:

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

System Interrupt Processing Summary

Referring to Figure 6-6 on page 6-24, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

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

3. The SIC_IMASKx register masks-off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IARx register, which maps the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7-IVG15), determines the core priority of interrupt A.
5. The ILAT bit adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. the IMASK bit masks-off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's ISR.
8. When the event vector for interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, the IPEND bit tracks all pending interrupts, as well as those being presently serviced.
9. When the interrupt service routine for interrupt A is executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISRx bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

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

If multiple interrupt sources share a single core interrupt, then the interrupt service routine (ISR) must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

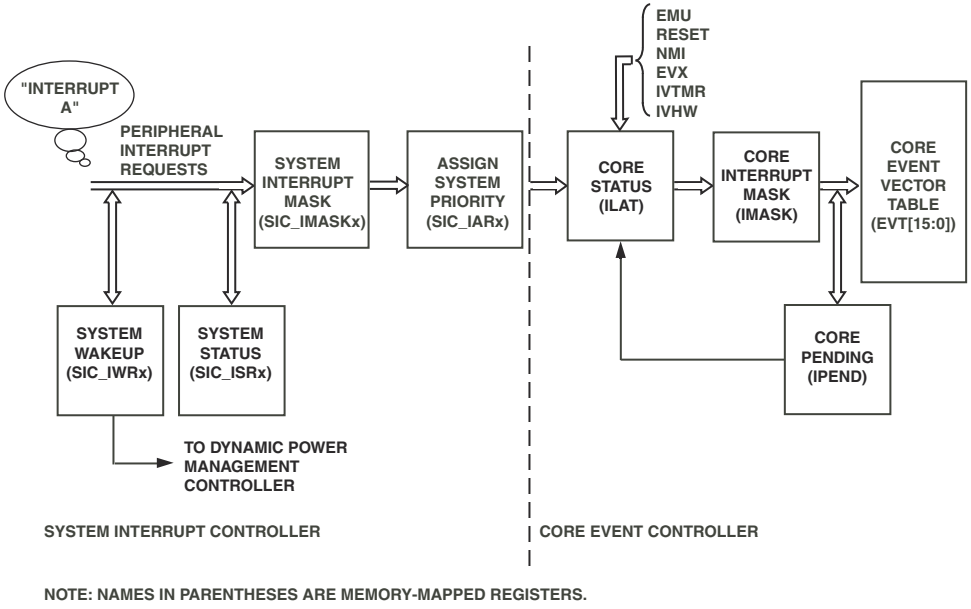


Figure 6-6. Interrupt Processing Block Diagram

System Interrupt Controller Registers

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

The SIC registers are described in Table 6-3.

Table 6-3. System Interrupt Controller Registers

Register Name	Description
SIC_IARx	"System Interrupt Assignment (SIC_IARx) Registers" on page 6-25
SIC_IMASKx	"System Interrupt Mask (SIC_IMASKx) Registers" on page 6-32
SIC_ISRx	"System Interrupt Status (SIC_ISRx) Registers" on page 6-35
SIC_IWRx	"System Interrupt Wakeup (SIC_IWRx) Registers" on page 6-37

System Interrupt Assignment (SIC_IARx) Registers

Table 6-4 defines the value to write in the `SIC_IARx` registers to configure a peripheral for a particular IVG priority.

Table 6-4. IVG Select Definitions

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

The system interrupt assignment registers (SIC_IARx) are shown in Figure 6-7 through Figure 6-18.

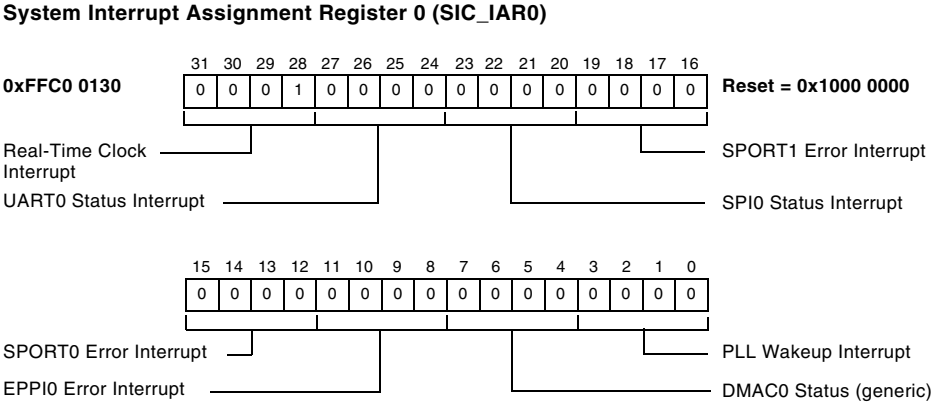


Figure 6-7. System Interrupt Assignment Register 0

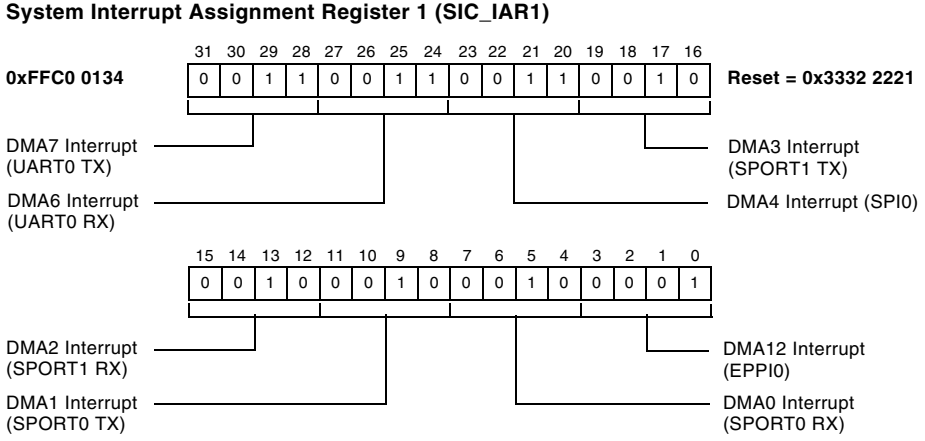


Figure 6-8. System Interrupt Assignment Register 1

System Interrupt Assignment Register 2 (SIC_IAR2)

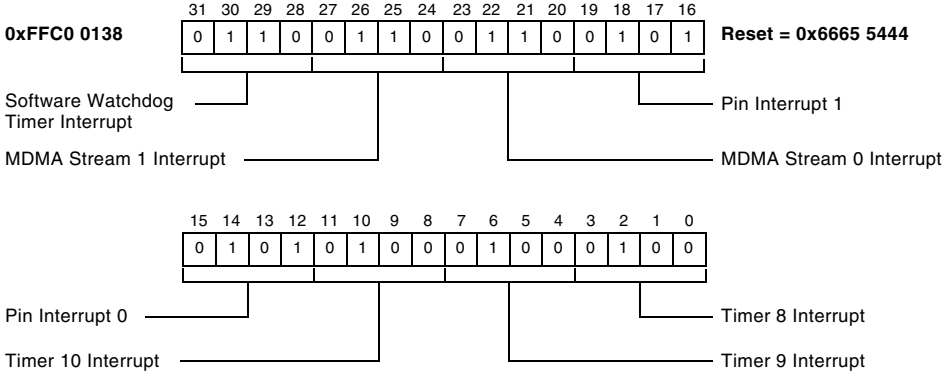


Figure 6-9. System Interrupt Assignment Register 2

System Interrupt Assignment Register 3 (SIC_IAR3)

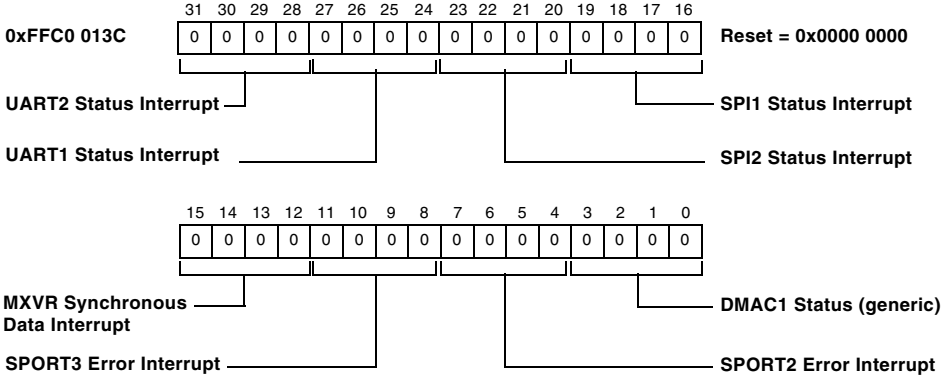


Figure 6-10. System Interrupt Assignment Register 3

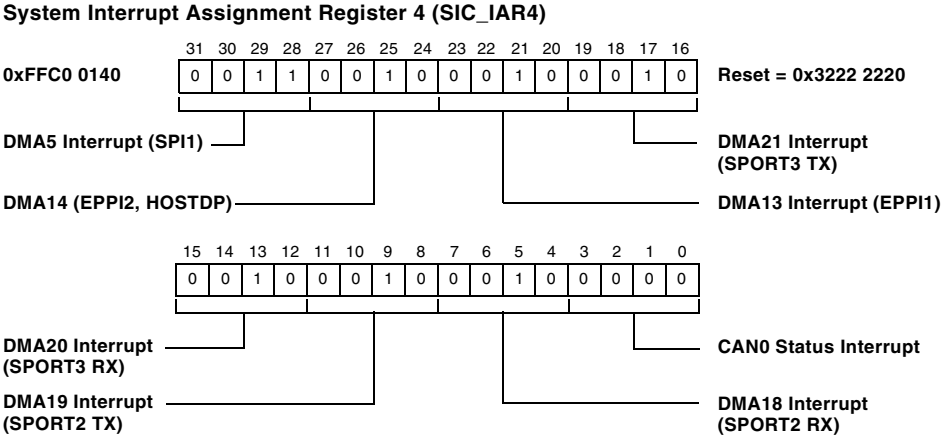


Figure 6-11. System Interrupt Assignment Register 4

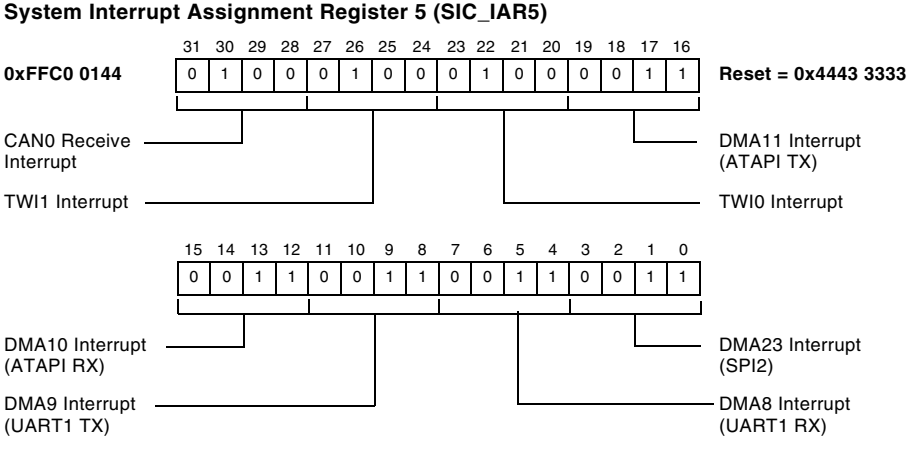


Figure 6-12. System Interrupt Assignment Register 5

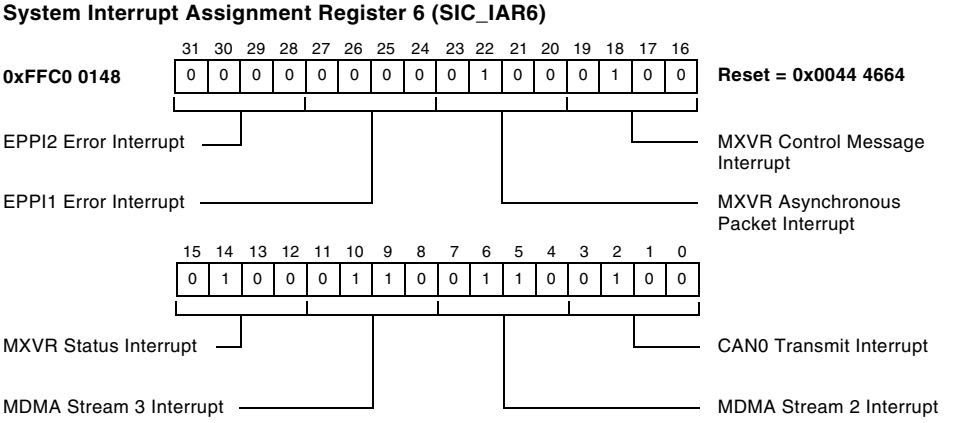


Figure 6-13. System Interrupt Assignment Register 6

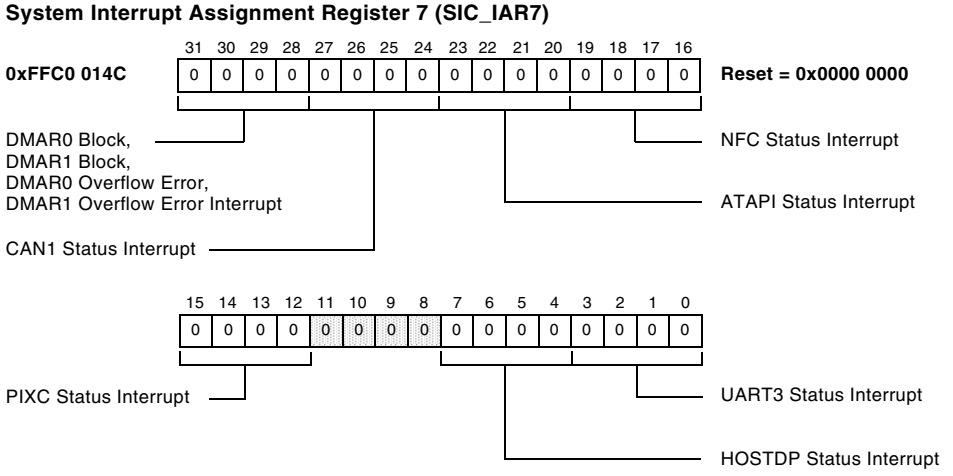


Figure 6-14. System Interrupt Assignment Register 7

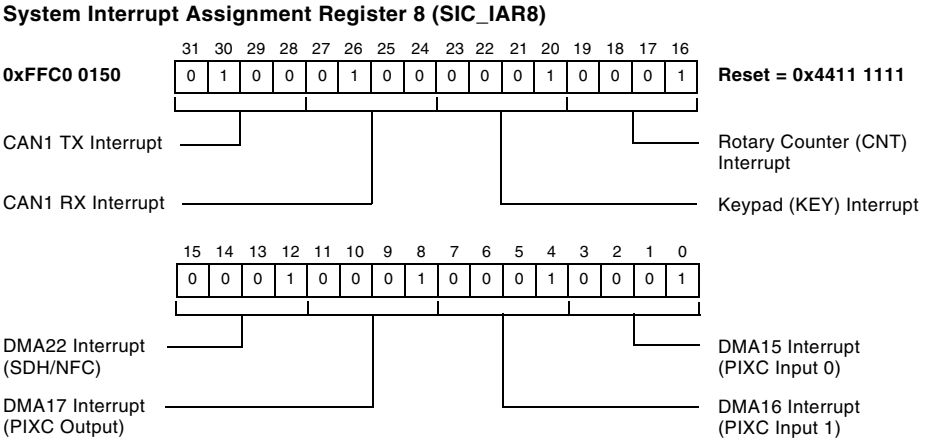


Figure 6-15. System Interrupt Assignment Register 8

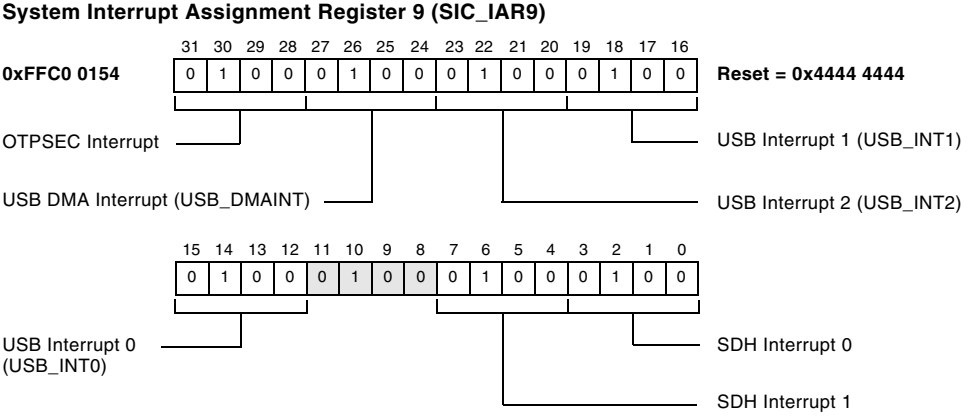


Figure 6-16. System Interrupt Assignment Register 9

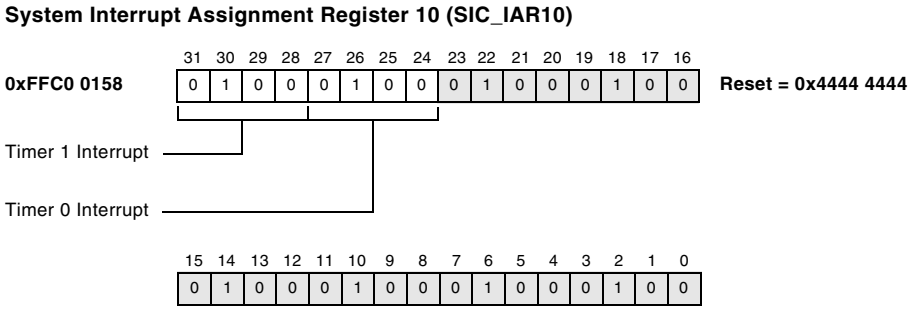


Figure 6-17. System Interrupt Assignment Register 10

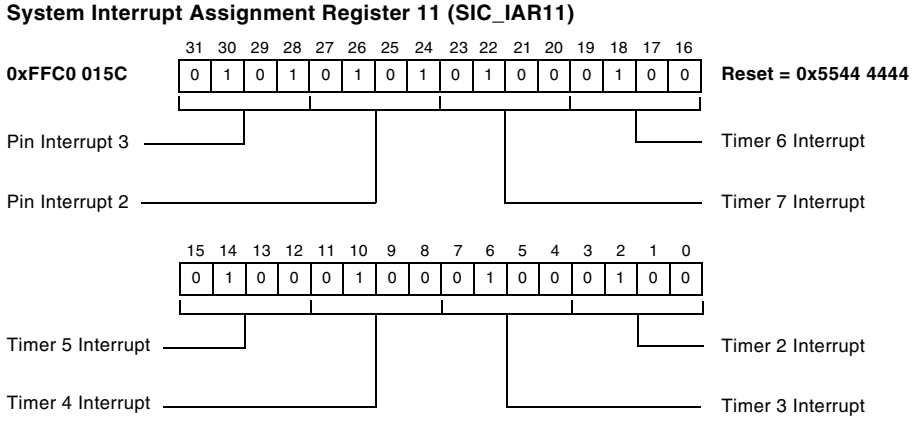


Figure 6-18. System Interrupt Assignment Register 11

System Interrupt Mask (SIC_IMASKx) Registers

The system interrupt mask registers (SIC_IMASKx) are shown in Figure 6-19 through Figure 6-21.

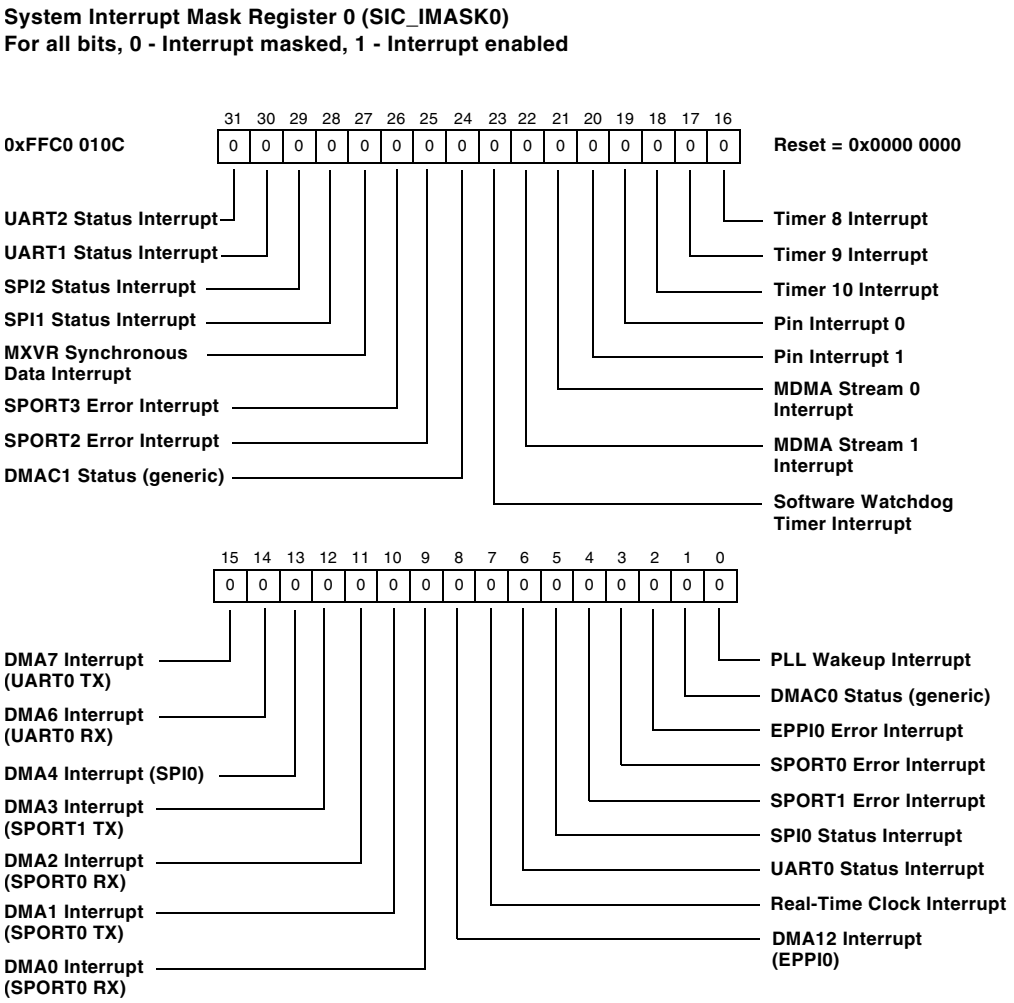


Figure 6-19. System Interrupt Mask Register 0

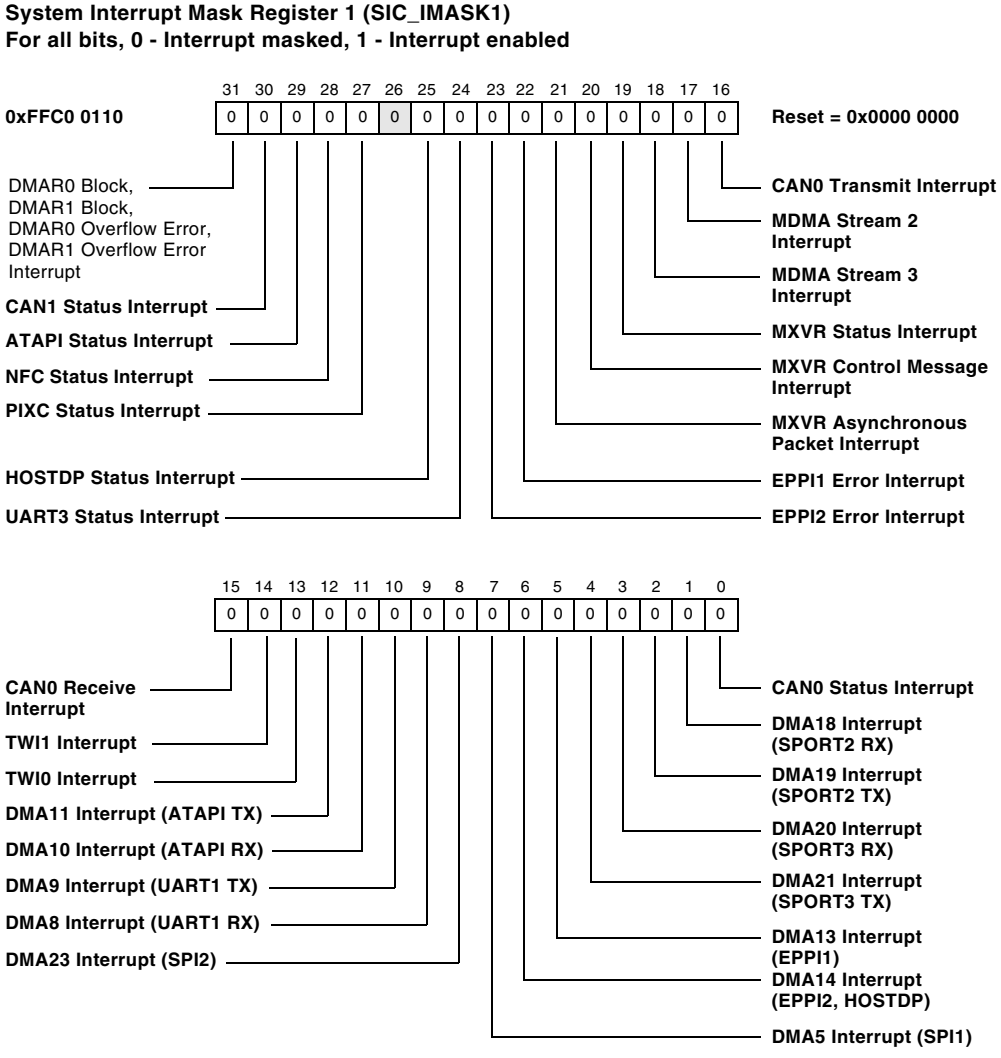


Figure 6-20. System Interrupt Mask Register 1

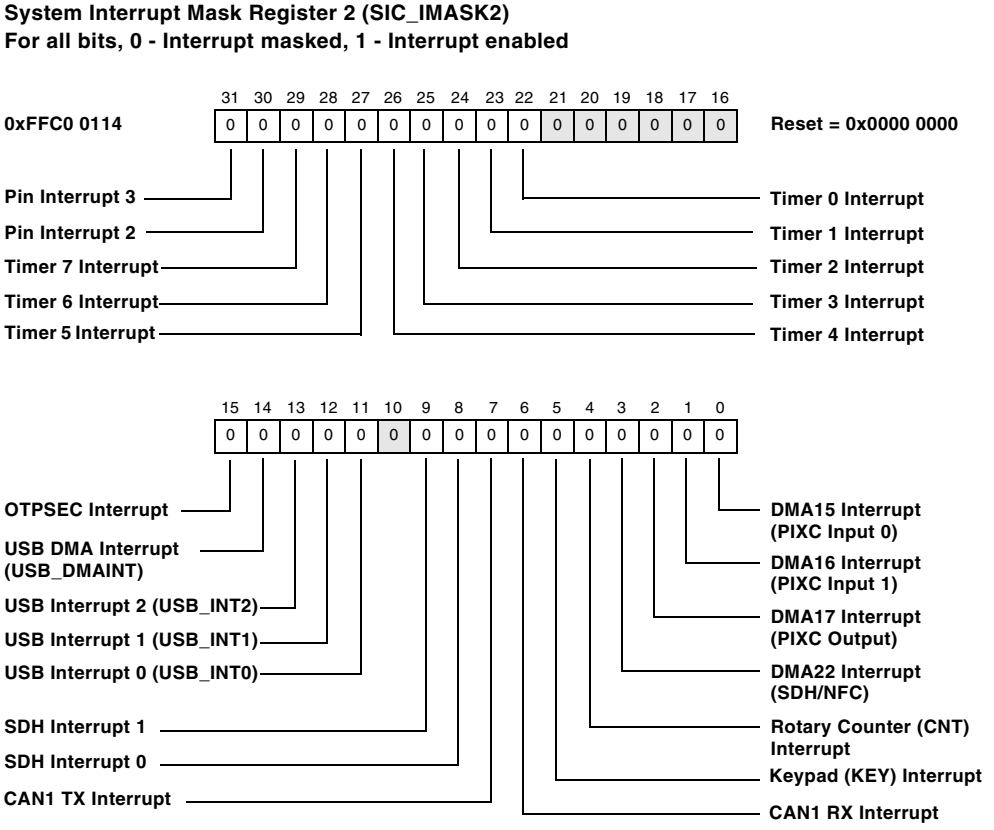


Figure 6-21. System Interrupt Mask Register 2

System Interrupt Status (SIC_ISRx) Registers

The system interrupt status registers (SIC_ISRx) are shown in Figure 6-22, Figure 6-23, and Figure 6-24.

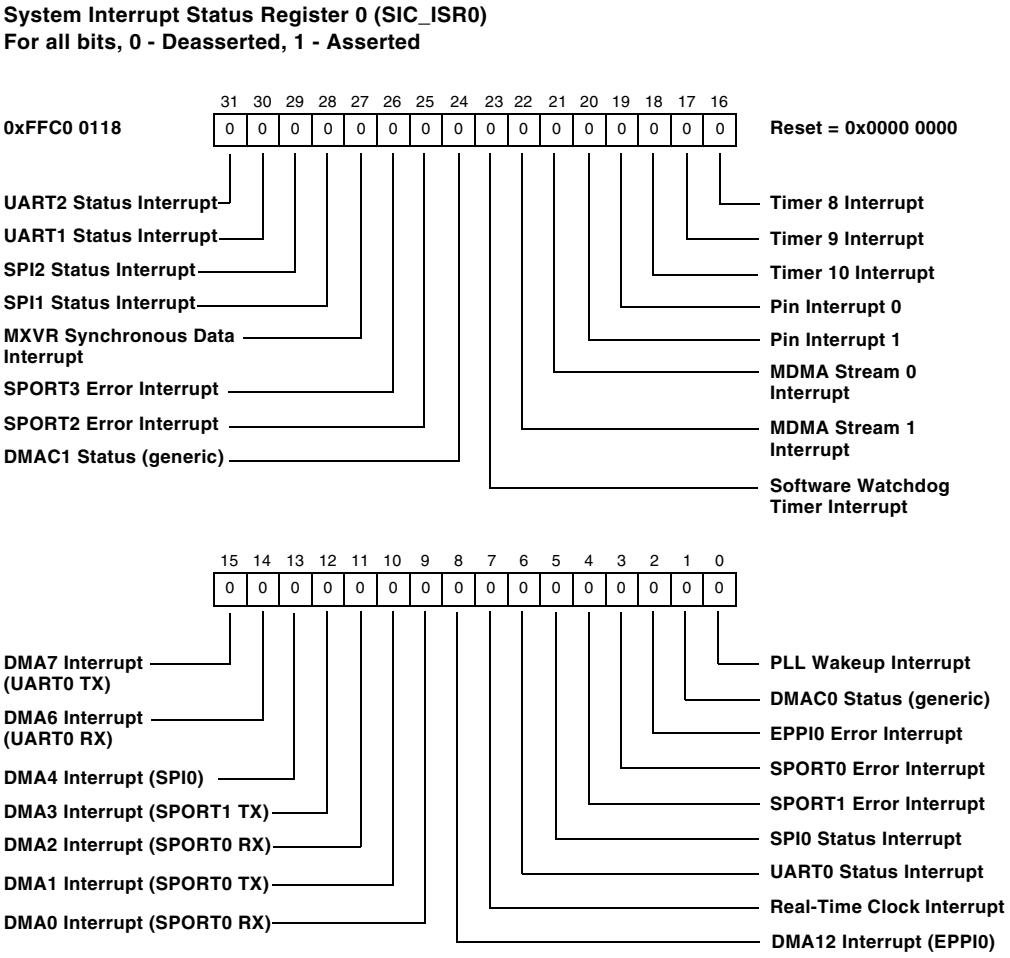


Figure 6-22. System Interrupt Status Register 0

System Interrupt Status Register 1 (SIC_ISR1)
For all bits, 0 - Deasserted, 1 - Asserted

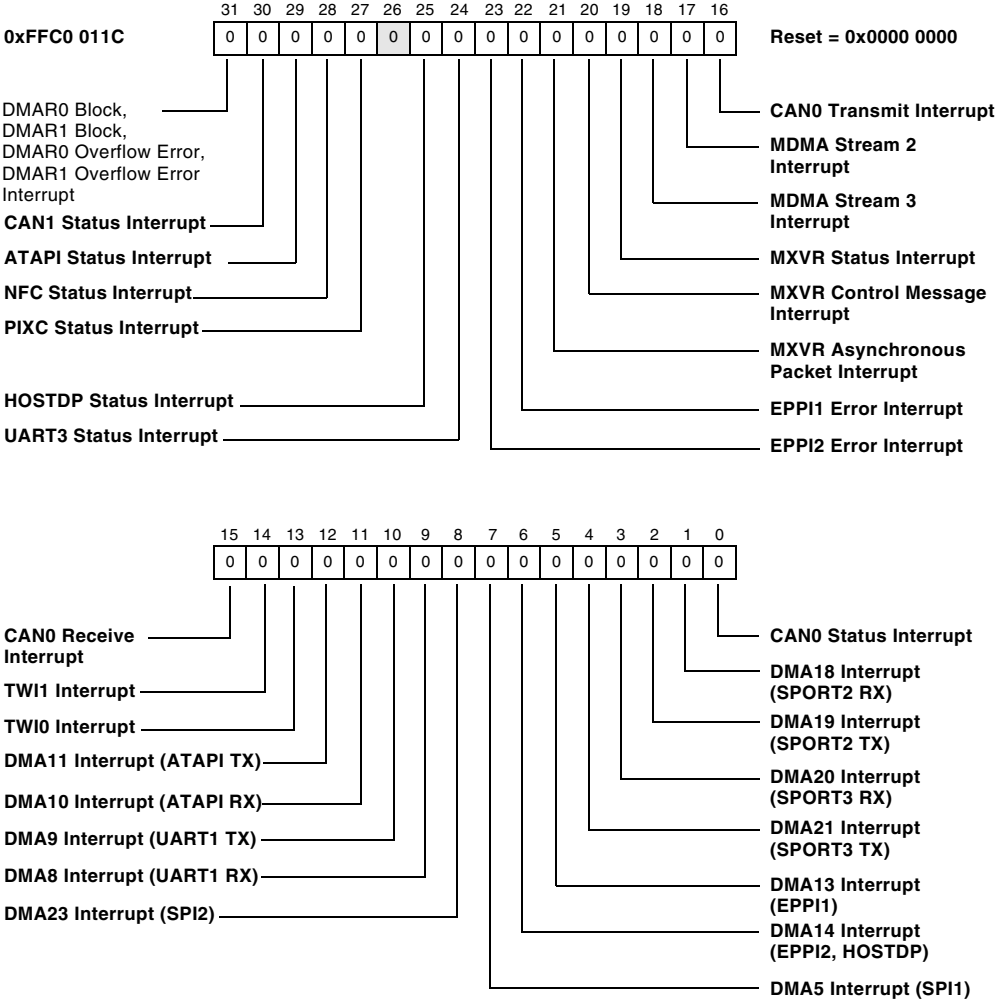


Figure 6-23. System Interrupt Status Register 1

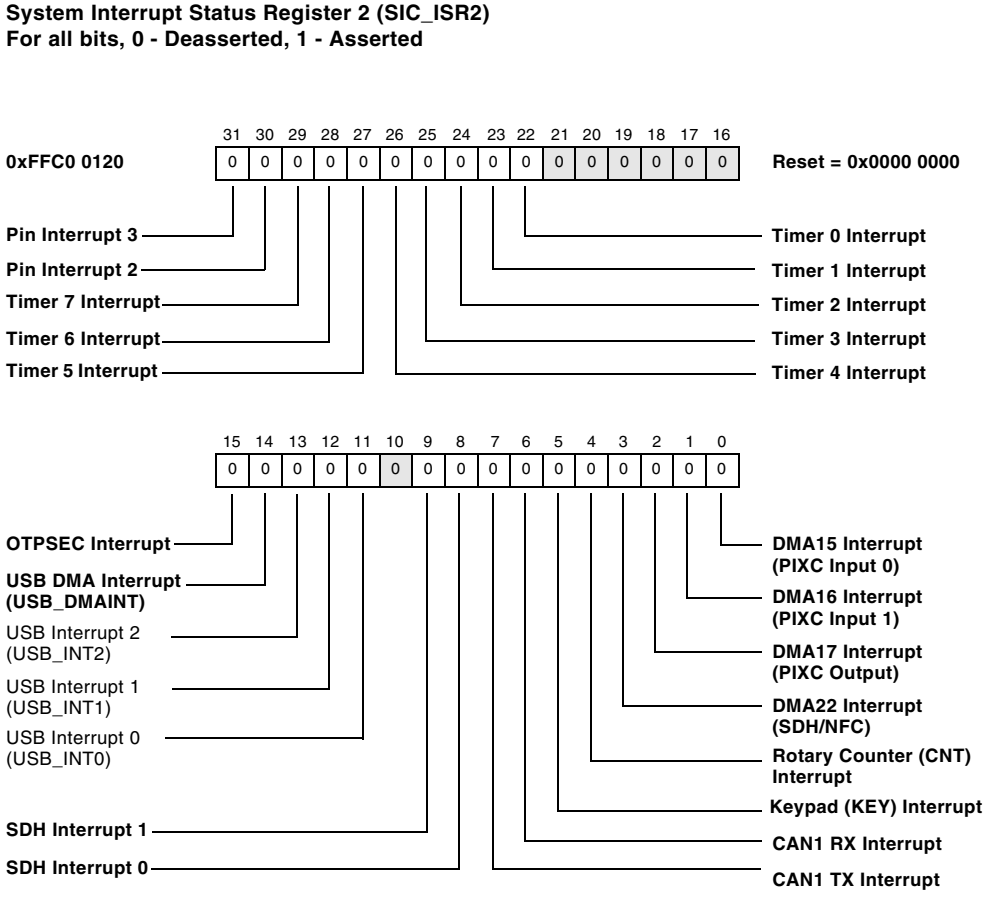


Figure 6-24. System Interrupt Status Register 2

System Interrupt Wakeup (SIC_IWRx) Registers

The system interrupt wakeup registers (`SIC_IWRx`) are shown in Figure 6-25, Figure 6-26, and Figure 6-27.

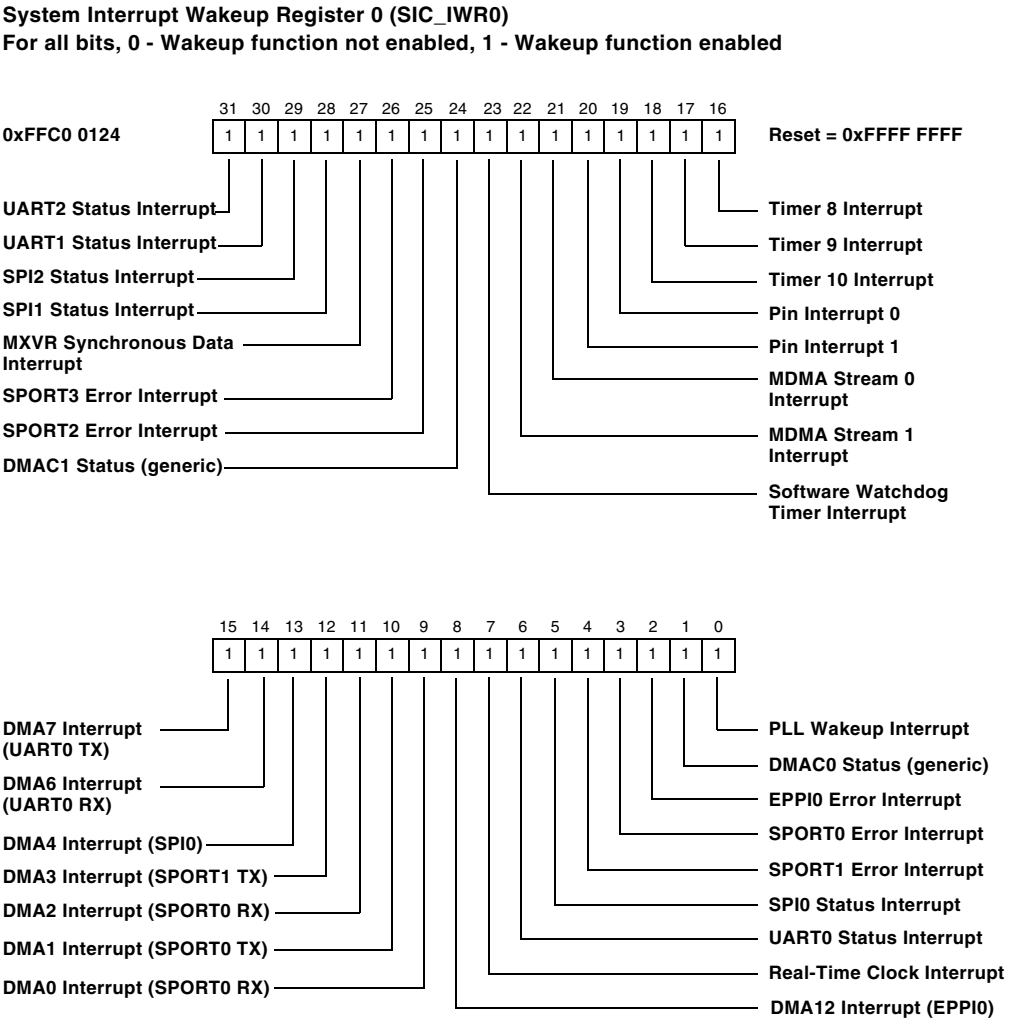


Figure 6-25. System Interrupt Wakeup Register 0

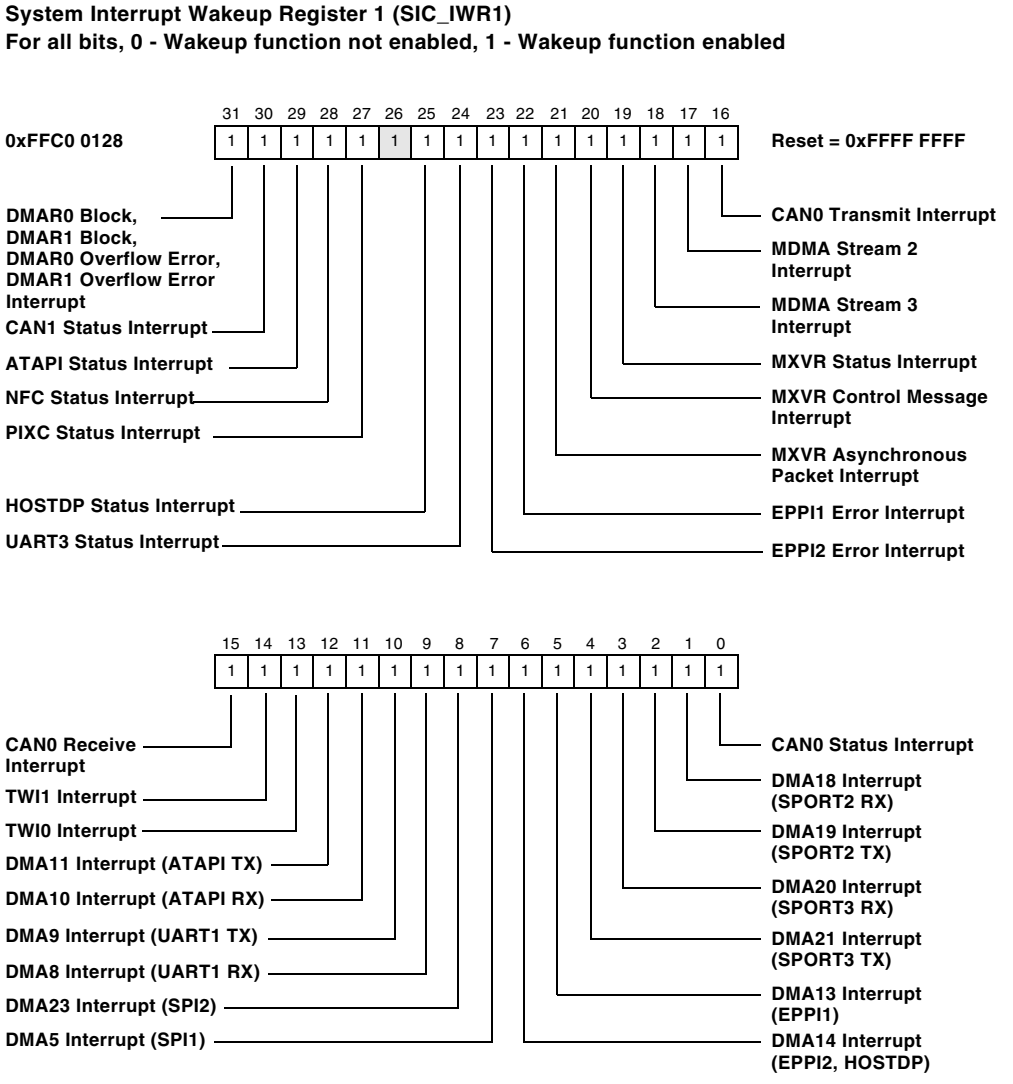


Figure 6-26. System Interrupt Wakeup Register 1

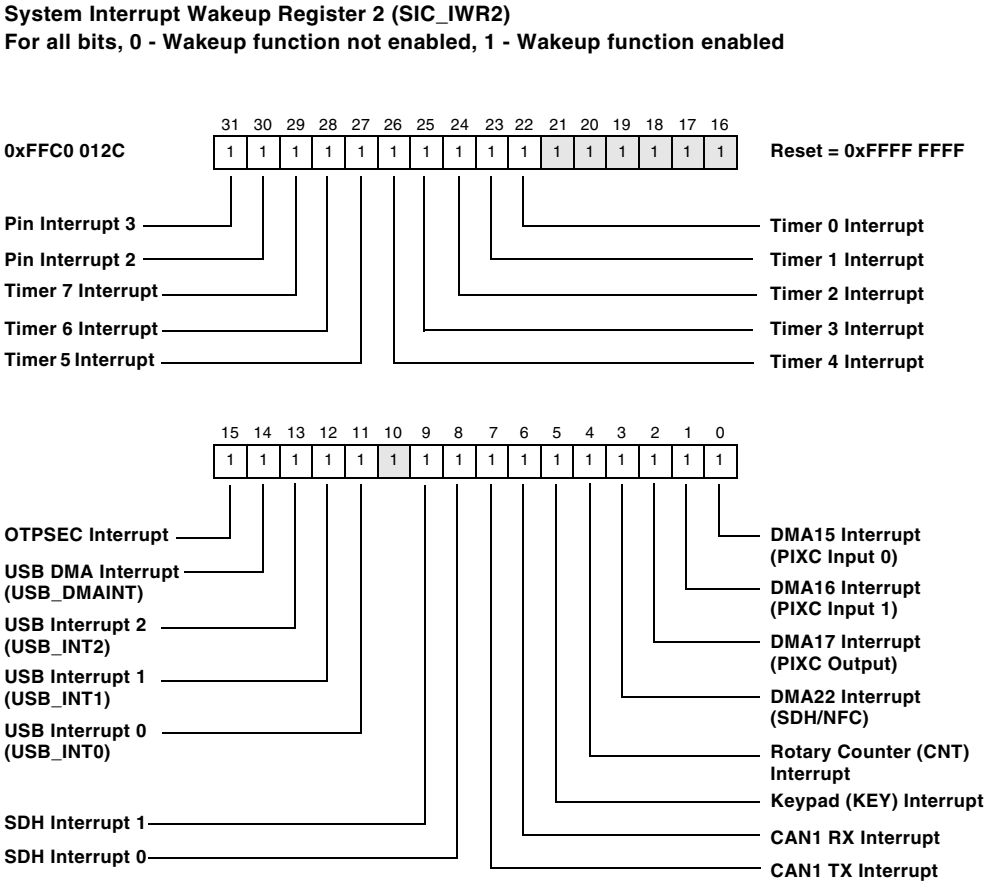


Figure 6-27. System Interrupt Wakeup Register 2

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 6-1 shows a representative example of how a GPIO interrupt request might be serviced.

Listing 6-1. Servicing GPIO Interrupt Request

```
#include <defBF549.h>
.section program;
_portg_a_isr:
  /* push used registers */
```

```
    [--sp] = (r7:7, p5:5);
    /* clear interrupt request on GPIO pin PG2 */
    /* no matter whether used A or B channel */
    p5.l = lo(PORTGIO_CLEAR);
    p5.h = hi(PORTGIO_CLEAR);
    r7 = PG2;
    w[p5] = r7;

    /*  place user code here */

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

The 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 6-1.

The level-sensitive nature of peripheral interrupts enables more than one of them to share the same IVG channel and therefore the same interrupt priority. This is programmable using the assignment registers. Then a

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

7 DIRECT MEMORY ACCESS

This chapter describes the direct memory access (DMA) controllers. The features common to all the DMA channels, as well as how DMA operations are set up are also described. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in "DAB, DCB, and DEB Performance" on page 2-22.

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

The chapter includes the following sections:

- "Overview and Features" on page 7-2
- "DMA Controller Overview" on page 7-6
- "Modes of Operation" on page 7-16
- "Functional Description" on page 7-25
- "Programming Model" on page 7-60
- "DMA Registers" on page 7-73
- "Programming Examples" on page 7-122

Overview and Features

The processor uses DMA to transfer data between memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity.

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

The DMA controllers can perform several types of data transfers:

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

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

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

DMA transfers on the processor can be descriptor-based or register-based. Register-based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed. Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes.

Examples of DMA styles supported by flex descriptors include:

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

- 1D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing a link pointer and a 32-bit address}
- 1D DMA, using a linked list of 5-word descriptors containing a link pointer, a 32-bit address, the length of the buffer, and the DMA configuration.
- 2D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page
- 2D DMA, using a linked list of 9-word descriptors, specifying everything

The following functions can be served by DMA channels:

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

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

DMA Controller Overview

Figure 7-1 and Figure 7-2 provide block diagrams of the DMA controllers.

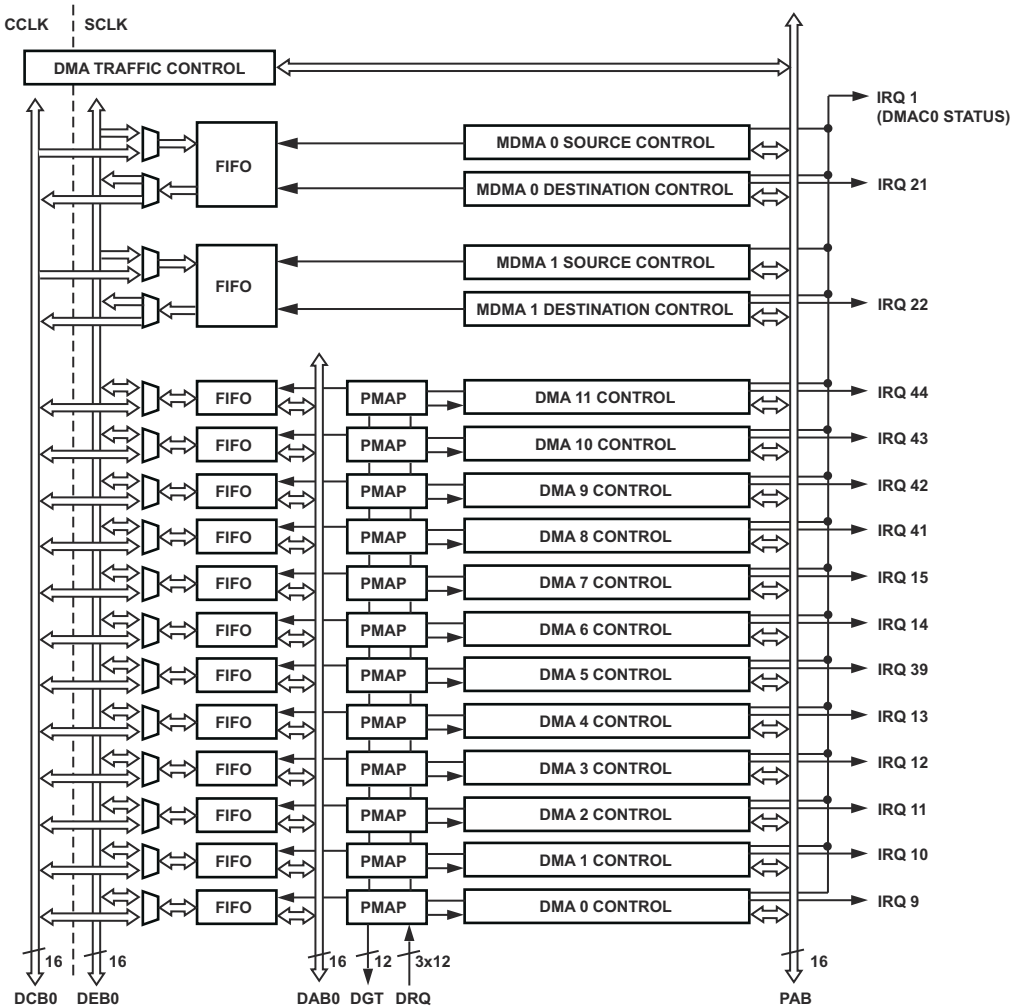


Figure 7-1. DMAC0 Controller Block Diagram

In the figures, DRQ = DMA request (see Table 7-21 on page 7-112) and DGT = DMA grant.

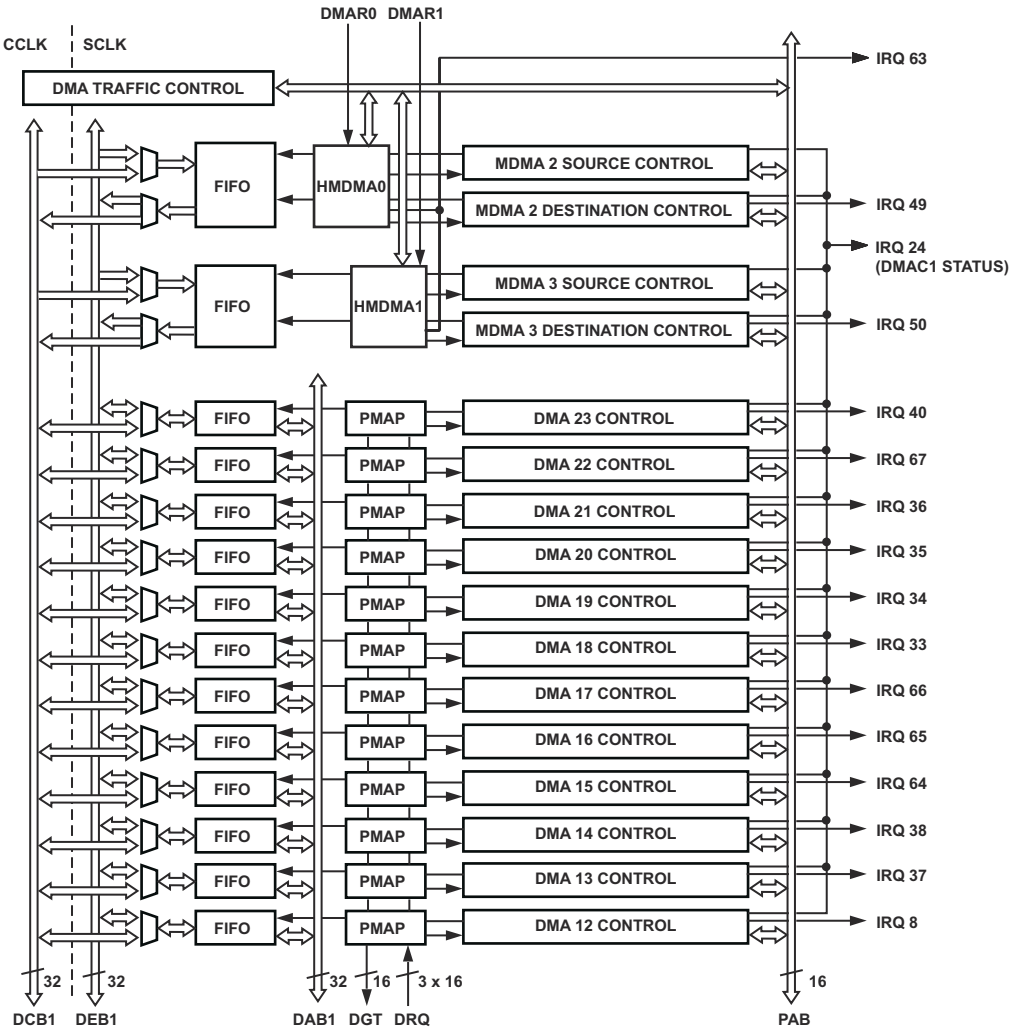


Figure 7-2. DMAC1 Controller Block Diagram

External Interfaces

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

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

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

Internal Interfaces

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

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

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

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

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

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

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

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

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

For more details on DMA performance see “DMA Performance” on page 7-50.

Peripheral DMA

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

Table 7-1. Default Mapping of Peripheral to DMA

DMA Channel	DMA Controller	PMAP Default[1, 2]	Peripheral Mapped by Default
DMA0	DMAC0	0x0	SPORT0 receive
DMA1	DMAC0	0x1	SPORT0 transmit
DMA2	DMAC0	0x2	SPORT1 receive
DMA3	DMAC0	0x3	SPORT1 transmit
DMA4	DMAC0	0x4	SPI0 receive/transmit
DMA5	DMAC0	0x5	SPI1 receive/transmit
DMA6	DMAC0	0x6	UART0 receive
DMA7	DMAC0	0x7	UART0 transmit
DMA8	DMAC0	0x8	UART1 receive
DMA9	DMAC0	0x9	UART1 transmit
DMA10	DMAC0	0xA	ATAPI receive
DMA11	DMAC0	0xB	ATAPI transmit
DMA12	DMAC1	0x0	EPPI0 receive/transmit
DMA13	DMAC1	0x1	EPPI1 receive/transmit
DMA14	DMAC1	0x2	EPPI2/Host DMA receive/transmit
DMA15	DMAC1	0x3	PIXC image data (read from memory)
DMA16	DMAC1	0x4	PIXC overlay data (read from memory)
DMA17	DMAC1	0x5	PIXC output data (write to memory)

Table 7-1. Default Mapping of Peripheral to DMA (Cont'd)

DMA Channel	DMA Controller	PMAP Default[1], [2]	Peripheral Mapped by Default
DMA18	DMAC1	0x6	SPORT2 receive
DMA19	DMAC1	0x7	SPORT2 transmit
DMA20	DMAC1	0x8	SPORT3 receive
DMA21	DMAC1	0x9	SPORT3 transmit
DMA22	DMAC1	0xA	SDH/NFC receive/transmit
DMA23	DMAC1	0xB	SPI2 receive/transmit
–	DMAC1	0xC	Note[3]
–	DMAC1	0xD	Note[3]
–	DMAC1	0xE	Note[3]
–	DMAC1	0xF	Note[3]

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

2 NFC (NAND flash controller) and SDH (secure digital host) share a PMAP assignment on DMAC1. For more information on enabling the NFC, see "DMA Controller 1 Peripheral Multiplexer (DMAC1_PERIMUX) Register" on page 7-121.

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

The default configuration works in most cases, but there are some cases where remapping the assignment can be helpful, because of the DMA channel priorities. In the default configuration, when competing for any of the system buses, DMA0 has higher priority than DMA1, and so on. DMA11 has the lowest priority of the peripheral DMA channels on DMAC0. Similarly, DMA12 is the highest priority peripheral DMA channel on DMAC1, and DMA23 is the lowest.

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

There are control bits in the SYSCR register which can change the priorities of DMAC0 and DMAC1 for L1 and for L2. For more information, see Table 2-1 on page 2-13 and Table 2-5 on page 2-21.

A 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field on DMAC0) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

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

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

Refer to the SYNC bit in the DMAx_CONFIG register for details (see "DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers" on page 7-79).

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

Memory DMA

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

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

Each MDMA controller provides two DMA channels:

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

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

- Priority 12: MDMA0 destination (DMAC0) or MDMA2 destination (DMAC1)
- Priority 13: MDMA0 source (DMAC0) or MDMA2 source (DMAC1)

- Priority 14: MDMA1 destination (DMAC0) or MDMA3 destination (DMAC1)
- Priority 15: MDMA1 source (DMAC0) or MDMA3 source (DMAC1)

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

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

The channels support 8-, 16-, and 32-bit memory DMA transfers, but both ends of MDMA0 and MDMA1 connect to 16-bit buses. Source and destination channel must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of the MDMA FIFO for a given stream are granted priority at the same time. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the external access bus (EAB) and DMA access bus (DAB) to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total transfer count is the same.

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

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

There are default sets of arbitration priorities between the different DMA controllers. These arbitration priorities are described in "DMA-Related Buses" on page 2-17.

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

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

In contrast, a memory DMA channel pair on a given controller can transfer data on every SCLK cycle if no other DMA activity occurs on the same DMA controller. This throughput difference can cause bandwidth issues with respect to other DMA controllers. For example, memory DMAs on the higher priority DMA controller will hold off transfers from peripherals on the lower priority DMA controller. This transfer holdoff is most apparent at the external memory interface.

To help with this scenario, refer to the arbitration options in "DMA-Related Buses" on page 2-17. Also refer to the descriptions of the DEB_ARB_PRIORITY, DEB0_URGENT, DEB1_URGENT, and DEB2_URGENT bits in the DDR_QUEUE register (see Table 5-4 on page 5-18) for additional control information.

Handshaked Memory DMA Mode

Handshaked operation applies only to memory DMA channels on DMAC1.

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

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

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

See “Handshaked Memory DMA Operation” on page 7-45 for a functional description.

Modes of Operation

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

Register-Based DMA Operation

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

For basic operation the software performs these steps:

- Write the source or destination address to the 32-bit DMAx_START_ADDR register.
- Write the number of data words to be transferred to the 16-bit DMAx_X_COUNT register.
- Write the address modifier to the 16-bit DMAx_X_MODIFY register. This is the two's-complement value added to the address pointer after every transfer. Typically, this register is set to 0x0004 for 32-bit DMA transfers, to 0x0002 for 16-bit transfers, and to 0x0001 for byte transfers.
- Write the operation mode to the DMAx_CONFIG register. These bits in particular need to be changed as needed:
    - The DMAEN bit enables the DMA channel.
    - The WNR bit controls the DMA direction. DMAs that read from memory keep this bit cleared, for example, transmitting peripheral DMAs and the source channel of memory DMAs. Receiving DMAs and the destination for memory DMAs set this bit, because they write to memory.
    - The WDSIZE bit controls the data word width for the transfer. It can be 8, 16, or 32 bits wide.

- The DI_EN bit enables an interrupt when the DMA operation has finished.
- Set the FLOW field to 0x0 for stop mode or 0x1 for autobuffer mode.

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

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

Stop Mode

In stop mode, the DMA operation is executed only once. If started, the DMA channel transfers the desired number of data words and stops itself again when finished. If the DMA channel is no longer used, software clears the DMAEN enable bit to disable a paused channel. Stop mode is entered if the FLOW bit field in the DMA channel's DMAx_CONFIG register is 0. The NDSIZE field must always be 0 in this mode.

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

Autobuffer Mode

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

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

Two-Dimensional DMA Operation

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

In two-dimensional (2D) mode the DMAx_X_COUNT register is accompanied by the DMAx_Y_COUNT register, supporting arbitrary row and column sizes up to 64K bytes x 64K bytes elements, as well as arbitrary DMAx_X_MODIFY and DMAx_Y_MODIFY values up to ±32K bytes. Furthermore, DMAx_Y_MODIFY values can be negative, allowing implementation of interleaved data streams. The DMAx_X_COUNT and DMAx_Y_COUNT values specify the row and column sizes, where a DMAx_X_COUNT value must be 2 or greater.

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

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

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

After the last transfer completes, registers DMAx_CURR_Y_COUNT = 1, DMAx_CURR_X_COUNT = 0, and DMAx_CURR_ADDR are equal to the last item's address plus DMAx_X_MODIFY. Note if the DMA channel is programmed to refresh automatically (autobuffer mode), then these registers is loaded from DMAx_X_COUNT, DMAx_Y_COUNT, and DMAx_START_ADDR upon the first data transfer.

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

Examples of Two-Dimensional DMA

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

```
DMAx_X_MODIFY = 1
DMAx_X_COUNT = 16
DMAx_Y_MODIFY = N-15 (offset from the end of one row to the start
of another)
DMAx_Y_COUNT = 8
```

This produces the following code offset from the start address:

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

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) × (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 code offset from the start address:

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

Descriptor-Based DMA Operation

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

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

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

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

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

Descriptor List Mode

Descriptor list mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to either 0x6 (small descriptor mode) or 0x7 (large descriptor mode). In this mode multiple descriptors form a chained list. Every descriptor contains a pointer to the next descriptor. When the descriptor is fetched, this pointer value is loaded into the DMAx_NEXT_DESC_PTR register of the DMA channel. In large descriptor mode this pointer is 32 bits wide. Therefore, the next descriptor may reside in any address space accessible through the DCB and DEB buses. In small descriptor mode this pointer is just 16 bits wide. For this reason, the next descriptor must reside in the same 64K byte address space as the first one, because the upper 16 bits of the DMAx_NEXT_DESC_PTR register are not updated.

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

Descriptor Array Mode

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

Variable Descriptor Size

In any descriptor-based mode, the NDSIZE field in the configuration word specifies how many 16-bit words of the next descriptor need to be loaded on the next fetch. In descriptor-based operation, NDSIZE field must be nonzero. The descriptor size can be any value from one entry (the lower 16 bits of DMAx_START_ADDR register only) to nine entries (all the DMA parameters). Table 7-2 illustrates how a descriptor must be structured in memory. The values have the same order as the corresponding MMR addresses.

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

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 7-2 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. For more information regarding descriptor element acronyms, refer to Table 7-5 on page 7-74.

Table 7-2. Parameter Registers and Descriptor Offsets

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

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

Mixing Flow Modes

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

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

Functional Description

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

DMA Operation Flow

Figure 7-3 and Figure 7-4 describe the DMA flow.

DMA Startup

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

Before initiating DMA for the first time on a given channel, be sure to initialize all parameter registers. Be especially careful to initialize the upper 16 bits of the DMAx_NEXT_DESC_PTR and DMAx_START_ADDR registers, because they might not otherwise be accessed, depending on the chosen FLOW mode of operation. Also note that the DMAx_X_MODIFY and DMAx_Y_MODIFY are not preset to a default value at reset.

To start DMA operation on a given channel, some or all of the DMA parameter registers must first be written directly. At a minimum, the DMAx_NEXT_DESC_PTR register (or DMAx_CURR_DESC_PTR register in FLOW = 4 mode) must be written at this stage, but the user may wish to write other DMA registers that might be static throughout the course of DMA activity (for example, DMAx_X_MODIFY and DMAx_Y_MODIFY). The contents of NDSIZE and FLOW in the DMAx_CONFIG register indicate which registers (if any) are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMAx_CONFIG with DMAEN = 1.

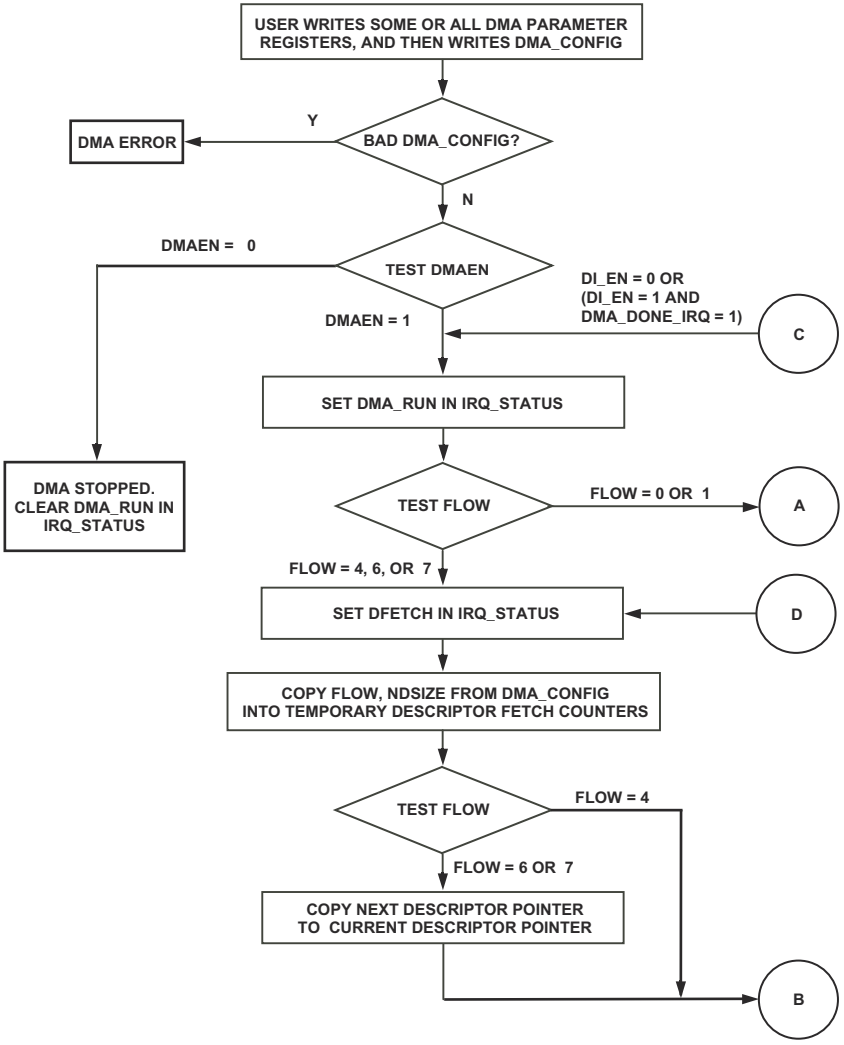


Figure 7-3. DMA Flow, From DMA Controller's Point of View (1 of 2)

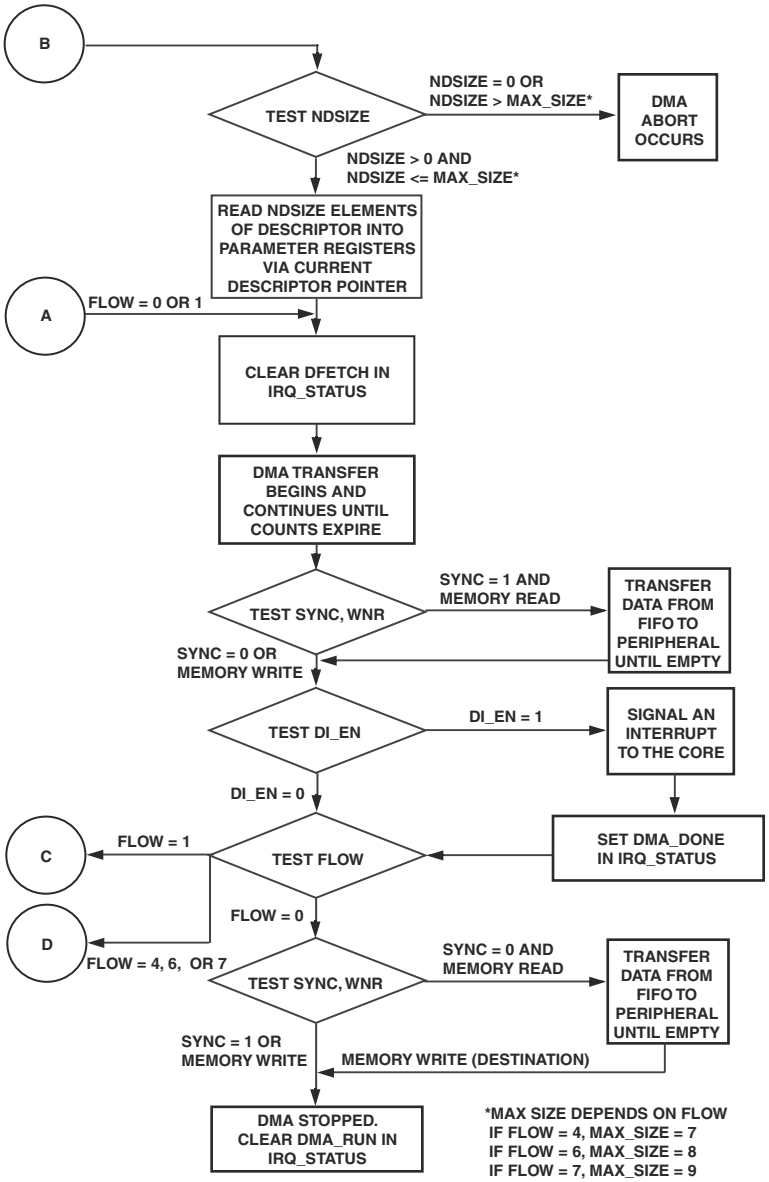


Figure 7-4. DMA Flow, From DMA Controller's Point of View (2 of 2)

When the DMAx_CONFIG register is written directly by software, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine is stopped (FLOW = 0).

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

As Figure 7-3 on page 7-26 and Figure 7-4 on page 7-27 show, at startup, the FLOW and NDSIZE bits in DMAx_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values.

If the FLOW value specifies small or large descriptor list modes, the DMAx_NEXT_DESC_PTR register is copied into DMAx_CURR_DESC_PTR register. Then, fetches of new descriptor elements from memory are performed, indexed by DMAx_CURR_DESC_PTR register, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into DMAx_NEXT_DESC_PTR register, but the fetch of the current descriptor continues using DMAx_CURR_DESC_PTR register. After completion of the descriptor fetch, DMAx_CURR_DESC_PTR register points to the next 16-bit word in memory past the end of the descriptor.

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

If DMACFG is not part of the descriptor, the previous DMAx_CONFIG settings (as written by MMR access at startup) control the work unit operation. If DMACFG register is part of the descriptor, then the DMAx_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor's DMACFG and by the parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the DMAx_IRQ_STATUS register indicate the state of the DMA channel. After a write to DMAx_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, DMA data transfer operation begins. The DMA channel immediately attempts to fill its FIFO, subject to channel priority—a memory write (RX) DMA channel begins accepting data from its peripheral, and a memory read (TX) DMA channel begins memory reads, provided the channel wins the grant for bus access.

When the DMA channel performs its first data memory access, its address and count computations take their input operands from the start registers (DMAx_START_ADDR, DMAx_X_COUNT, DMAx_Y_COUNT), and write results back to the current registers (DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, DMAx_CURR_Y_COUNT). Note also that the current registers are not valid until the first memory access is performed, which may be some time after the channel is started by the write to the 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 7-4 on page 7-27.

DMA Refresh

On completion of a work unit, the DMA controller:

- Completes the transfer of all data between memory and the DMA unit.
- If SYNC = 1 and WNR = 0 (memory read). Selects a synchronized transition. Transfers all data to the peripheral before continuing.
- If enabled by DI_EN, signals an interrupt to the core and sets the DMA_DONE bit in the channel's DMAx_IRQ_STATUS register.
- If FLOW = 0 (stop) only. Stops operation by clearing the DMA_RUN bit in DMAx_IRQ_STATUS after any data in the channel's DMA FIFO is transferred to the peripheral.
- During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in DMAx_IRQ_STATUS to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

  If FLOW = 4 (descriptor array), then loads a new descriptor from memory into DMA registers by way of the contents of DMAx_CURR_DESC_PTR, while incrementing DMAx_CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMAx_CONFIG value prior to the beginning of the fetch.

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

by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR will retain their former value. This supports a shorter, more efficient descriptor than the descriptor list large model, suitable whenever the application can place the channel's descriptors in the same 64K byte range of memory.

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

Note if it is necessary to link from a descriptor chain whose descriptors are in one 64K byte area to another chain whose descriptors are outside that area, only one descriptor needs to use FLOW = 7—just the descriptor which contains the link leaving the 64K byte range. All the other descriptors located together in the same 64K byte areas may use FLOW = 6.

- If FLOW = 4, 6, or 7 (descriptor array, descriptor list small, or descriptor list large, respectively), then the DMA controller clears the DFETCH bit in the DMAx_IRQ_STATUS register.

- If FLOW = any value but 0 (Stop), then the DMA controller begins the next work unit, contending with other channels for priority on the memory buses. On the first memory transfer of the new work unit, the DMA controller updates the current registers from the start registers:

  DMAx_CURR_ADDR loaded from DMAx_START_ADDR
  DMAx_CURR_X_COUNT loaded from DMAx_X_COUNT
  DMAx_CURR_Y_COUNT loaded from DMAx_Y_COUNT

  The DFETCH bit in DMAx_IRQ_STATUS is then cleared, after which the DMA transfer begins again, as shown in Figure 7-4 on page 7-27.

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 versus external) as the current descriptor.

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

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

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

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

DMA Receive

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

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

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

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

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

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

DMA word size must not change between one descriptor and the next in any DMA receive (memory write) channel within a single descriptor chain, regardless of the SYNC bit setting. In other words, if a descriptor has WNR = 1 and FLOW = 4, 6, or 7, then the next descriptor must have the same word size. For any DMA receive (memory write) channel, there is no restriction on changes of memory space (internal versus external) between descriptors or descriptor chains. DMA transmit (memory read) channels may have such restrictions (see "DMA Transmit and MDMA Source" on page 7-33).

Stopping DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete. If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMAEN bit is 0. A write of 0 to the entire register always terminates DMA gracefully (without DMA abort).

If a channel is stopped abruptly by writing DMAx_CONFIG to 0 (or any value with DMAEN = 0), the user must ensure memory read or write accesses in the pipelines are complete before reenabling the channel. If the channel is reenabled 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 controllers flag conditions that cause DMA processes to end abnormally (that is, abort). This functionality is provided as a tool for system development and debug, as a way to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

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

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

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

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

- The configuration register contains invalid values:

  - Incorrect WDSIZE value (WDSIZE = b#11)

  - Bit 15 not set to 0

  - Incorrect FLOW value (FLOW = 2, 3, or 5)

- NDSIZE value does not agree with FLOW.
See Table 7-3 on page 7-39.

- 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 b#00.
- A memory space transition was attempted (internal-to-external or vice versa).
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (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 7-3. 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 on the processor, such as the Host DMA port are capable of managing some of their own DMA operations, thus dramatically improving real-time performance and relieving control and interrupt demands on the Blackfin processor core. These peripherals may communicate to the DMA controllers using DMA control commands, which are transmitted from the peripheral to the associated DMA channel over internal DMA request buses. These request buses consist of three wires per DMA-management-capable peripheral. The DMA control commands extend the set of operations available to the peripheral beyond the simple “request data” command used by peripherals in general. Refer to the appropriate peripheral chapter for a description on how that peripheral uses DMA control commands.

Note that while these DMA control commands are not visible to or controlled by the user, their use by a peripheral has implications for the structure of the DMA transfers which that peripheral can support. It is important that application software be written to comply with certain

restrictions regarding work units and descriptor chains (described later in this section) so that the peripheral operates properly whenever it issues DMA control commands.

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

The DMA control commands are shown in Table 7-4.

Table 7-4. DMA Control Commands

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

Additional information for the control commands includes:

- **Restart**

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

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

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

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

- **Finish**

  The finish control command causes the current work unit to terminate processing and move on to the next. An interrupt is signalled as usual, if selected by the `DI_EN` bit. The peripheral can thus use the finish command to partition the DMA stream into work units on its own, perhaps as a result of parsing the data currently passing 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 are flushed from the channel's pipelines, the channel signals an interrupt (if enabled), and begins fetching the next descriptor (if any). DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO.

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

- **Interrupt**

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

- **Request Data**

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

- **Request Data Urgent**

  The request data urgent control command behaves identically to the DMA request control command, except that while it is asserted the DMA channel performs its memory accesses with urgent priority. This includes both data and descriptor-fetch memory accesses.

A DMA-management-capable peripheral might use this control command if an internal FIFO is approaching a critical condition, for example.

- **Request Register Load**

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

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

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

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

Restrictions

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

Transmit Restart or Finish

No restart or finish control command may be issued by a peripheral to a channel configured for memory read unless both (a) the peripheral has already performed at least one DMA transfer in the current work unit, and (b) the current work unit has more than four items remaining in 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 five more than the maximum data count prior to any restart or finish command, the above restriction is satisfied. This implies that any work unit which might be managed by restart or finish commands must have DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing at least five 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 operates properly for 1D work units up to 65,531 data items or 2D work units up to 4,294,967,291 data items.

Receive Restart or Finish

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

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

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

Handshaked Memory DMA Operation

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

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

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

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

The HMDMAx_BCINIT registers control how many data transfers are performed upon every DMA request. If set to 1, the peripheral can time every individual data transfer. If greater than 1, the peripheral must feature sufficient buffer size to provide or consume the number of words programmed. Once the transfer is requested, no further handshake can hold off the DMA from transferring the entire block, except by stalling the EBIU accesses by the ARDY signal or a complete bus request and grant cycle through the $\overline{BR}$ 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 is started, the HMDMAx_BCOUNT registers return the remaining number of transfers to complete the current block. When the complete block is processed, the HMDMAx_BCOUNT register returns zero. Software can force a reload of the HMDMAx_BCOUNT from the HMDMAx_BCINIT register even during normal operation by writing a 1 to the RBC bit in the HMDMAx_CONTROL register. Set RBC only when the HMDMA module is already active, but the MDMA is not enabled.

Pipelining DMA Requests

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

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

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

Figure 7-5 illustrates how an asynchronous FIFO could be connected. In such a scenario, the REP bit is cleared to let the DMARx request pin listen to falling edges. The Blackfin processor does not evaluate the full flag such FIFOs usually provide, because asynchronous polling of that signal reduces the system throughput drastically. Moreover, the processor first fills the FIFO by initializing the HMDMAx_ECINIT register by the value 1024 which equals the depth of the FIFO. Once enabled, the MDMA automatically transmits 1024 data words. Afterward it continues to transmit only if the FIFO is emptied by its read strobe again.

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

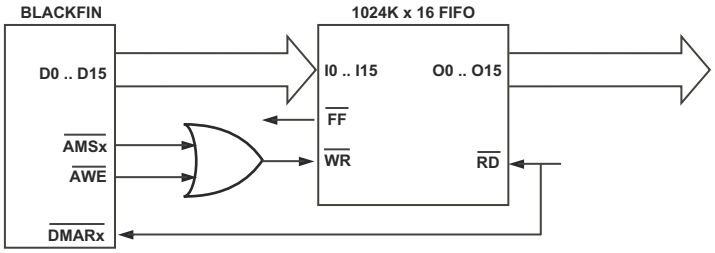


Figure 7-5. Transmit DMA Example Connection

In the receive example shown in Figure 7-6, the Blackfin processor again does not use the FIFO's internal control mechanism. Rather than testing the empty flag, the processor counts the number of data words available in the FIFO by its own HMDMAx_ECOUNT register. Theoretically, the MDMA could immediately process data as soon as it is written into the FIFO by the write strobe, but the fast MDMA engine would read out the FIFO quickly and stall soon if the FIFO was not filled with new data promptly. Streaming applications can balance the FIFO so that the producer is never held off by a full FIFO and the consumer is never held by an empty FIFO. This is accomplished by filling the FIFO half way and then letting both consumer and producer run at the same speed. In this case, the HMDMAx_ECINIT register can be written with a negative value, which corresponds to half the FIFO depth. Then, the MDMA does not start consuming data as long as the FIFO is not half filled.

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

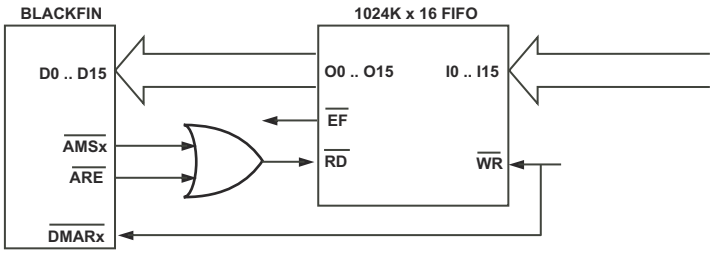


Figure 7-6. Receive DMA Example Connection

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

HMDMA Interrupts

In addition to the normal MDMA interrupt channels, the handshake hardware provides two new interrupt sources for each DMARx input. All interrupt sources (DMAR0 and DMAR1 block done, DMAR0 and DMAR1 overflow error) are routed to Peripheral Interrupt ID#63. Refer to Chapter 6, "System Interrupts" for more information. The HMDMAx_CONTROL registers provide interrupt enable and status bits. The interrupt status bits require a write-1-to-clear operation to cancel the interrupt request.

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

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

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

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

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

Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

DMA Throughput

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

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

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

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

Some considerations which limit the actual performance include:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example, for accessing the same L1 bank, for opening/closing DDR SDRAM pages, or while filling cache lines.
- Direction change from receive to transmit on the DAB bus imposes a one SCLK cycle delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to DMA registers other than DMAx_CONFIG, DMAx_IRQ_STATUS, or DMAx_PERIPHERAL_MAP stalls all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the control/status registers do not cause stalls or wait states.
- Reads from DMA registers other than control/status registers use one PAB bus wait state, delaying the core for several core clocks.

- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMAEN changes from 0 to 1 or when the SYNC bit is set to 1 in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

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

The MDMA controllers are clocked by SCLK. If source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. At a clock ratio of 2:1, for example, DMA typically runs at 2/3 of the system clock rate. At higher clock ratios, full bandwidth is maintained.

If source and destination are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to 3 plus the memory latency at the DMA in SCLKs (which is typically 7 for internal transfers and 6 for external transfers).

Memory DMA Timing Details

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

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

Static Channel Prioritization

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the PMAP field in the DMAx_PERIPHERAL_MAP registers. The memory DMA streams are always lower static priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers. Refer to Table 7-1 for detailed information on priority and mapping of peripherals to DMA.

Temporary DMA Urgency

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals is not too large a fraction of the total, then all peripherals' requests should be granted as required.

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

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

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

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

For DEB bus transfers, all DMA requests to the DDR controller can be marked "urgent" under software control by setting the corresponding DEBx_URGENT bit in the DDR_QUEUE register. Refer to "DDR Arbitration" on page 5-11 for more details.

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

DMA requests from an MDMA channel become urgent when handshaked operation is enabled and the DMARx edge count exceeds the value stored in the HMDMAx_ECURGENT register. If handshaked operation is disabled, software can control urgency of requests directly by altering the DRQ bit field in the HMDMAx_CONTROL register.

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

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only urgent request are granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1, L2, or external), and so are all prior incomplete memory transfers ahead of it in that memory system. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

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

Memory DMA Priority and Scheduling

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

MDMA0 and MDMA1 are always the lowest priority channels in DMAC0, but they have higher priority than all DMAC1 channels by default. Therefore, it is preferable to use MDMA2 and MDMA3 to avoid starving memory bandwidth to DMAC1 peripherals. Refer to "DCB Arbitration" on page 2-20 for a discussion of switching the relative priorities of DMAC0 and DMAC1.

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

By default, when more than one MDMA stream is enabled and ready, only the highest priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, however, the MDMA_ROUND_ROBIN_PERIOD register may be programmed to select each stream in turn for a fixed number of transfers.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the MDMA_ROUND_ROBIN_PERIOD field in the DMACx_TCPER register (see "Static Channel Prioritization" on page 7-54).

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 is granted (stream 0 in case of conflict), and that stream's selection is then "locked." The MDMA_ROUND_ROBIN_COUNT counter field in the DMACx_TCCNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of 1, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and on the next cycle the stream selection unlocks and becomes free again.

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

Traffic Control

In the Blackfin DMA architecture, there are two completely separate but simultaneous prioritization processes—the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA through the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this

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 7-54.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same direction transfers together. The DMA block provides a traffic control mechanism controlled by the DMACx_TCPER and DMACx_TCCNT registers. This mechanism performs the optimization without real-time processor intervention, and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMACx_TCCNT register. See "Memory DMA Priority and Scheduling" on page 7-56.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out, or until traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower priority channel 5 is going "with traffic" and higher priority channel 3 is going "against traffic," then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If, on the next cycle, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both "against traffic," then their effective priorities would become 19 and

22, respectively. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required by the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the `DMACx_TCPER` register to 0x0000.

Programming Model

Several synchronization and control methods are available for use in development of software tasks which manage peripheral DMA and memory DMA (see also "Memory DMA" on page 7-13). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each peripheral DMA and memory DMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the `DMAx_IRQ_STATUS` register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of DMAx_IRQ_STATUS, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note, since every interrupt channel has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Polling of the DMAx_CURR_ADDR, DMAx_CURR_DESC_PTR, or DMAx_CURR_X_COUNT/DMAx_CURR_Y_COUNT registers is not recommended as a method of precisely synchronizing DMA with data processing, due to DMA FIFOs and DMA/memory pipelining. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation are first visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the DDR SDRAM to perform a page open operation which takes many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but is stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by channel B's DMAx_CURR_ADDR has or has not been written, based on examination of the DMAx_CURR_ADDR register contents.

Polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software, however, if allowances are made for the lengths of the DMA/memory pipeline. The length of the DMA FIFO for a peripheral DMA channel is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) and the length for an MDMA FIFO is eight locations (four 32-bit data elements). The DMA does not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and external bus interface unit (EBIU) pipelines is approximately three data elements, when measured from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. (Note this is a maximum, as the DMA/memory pipeline may include traffic from other DMA channels.)

For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its `DMAx_CURR_X_COUNT` register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. The total pipeline length is no greater than the sum of 4 (for the peripheral DMA FIFO) plus 6 (for the DMA/memory pipeline), or 10 data elements, so it is safe to conclude that the DMA transfer of the first 40-10 = 30 data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's `DMAx_IRQ_STATUS` register to confirm completion of DMA, rather than polling current address/pointer/count registers. When the DMA system issues an interrupt or changes an `DMAx_IRQ_STATUS` bit, it guarantees that the last memory operation of the

work unit is complete and is visible to DSP code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO; for memory write DMA, the DMA unit will have received an acknowledge from L1 or memory other than L1 or the EBIU that the data is written.

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

Single-Buffer DMA Transfers

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

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

Continuous Transfers Using Autobuffering

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

- 1D, interrupt-driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
- 2D, interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set DI_SEL = 1 in DMAx_CONFIG) to signal at the completion of each DMA inner loop. In this way, a traditional double buffer or "ping-pong" scheme could be implemented.

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

  DMAx_START_ADDR = buffer base address
  DMAx_CONFIG = 0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1, DMA2D = 1, WDSIZE = 01, WNR = 1, DMAEN = 1)
  DMAx_X_COUNT = 512
  DMAx_X_MODIFY = 2 for 16-bit data
  DMAx_Y_COUNT = 2 for two sub-buffers
  DMAx_Y_MODIFY = 2, same as DMAx_X_MODIFY for contiguous sub-buffers

- 2D, polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2D multibuffer synchronization scheme may be used. For example,

assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

DMAx_START_ADDR = buffer base address
DMAx_CONFIG = 0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1, WDSIZE = 10, WNR = 1, DMAEN = 1)
DMAx_X_COUNT = 16
DMAx_X_MODIFY = 4 for 32-bit data
DMAx_Y_COUNT = 4 for four sub-buffers
DMAx_Y_MODIFY = 4, same as DMAx_X_MODIFY for contiguous sub-buffers

- The synchronization core might read DMAx_Y_COUNT to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of DMAx_Y_COUNT shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.
- 1D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1D autobuffer mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1D or 2D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory pool allocator, and a small trailer containing a

checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list as desired by selecting the appropriate FLOW setting in DMAx_CONFIG.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

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

- The lower byte of DMAx_CONFIG specifies the DMA transfer to be performed by the *current* descriptor (for example, interrupt-enable, 2D mode).
- The upper byte of DMAx_CONFIG specifies the format of the *next* descriptor in the chain. The NDSIZE and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is restarted, both bytes of the DMAx_CONFIG value written to the DMA channel's DMAx_CONFIG register should correspond to the current descriptor.

At a minimum, the FLOW, NDSIZE, WNR, and DMAEN fields must all agree with the current descriptor; the WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields are taken from the DMAx_CONFIG value in the descriptor read from memory (and the field values initially written to the register are ignored). See "Initializing Descriptors in Memory" on page 7-126 in the "Programming Examples" section for information on how descriptors can be set up.

Descriptor Queue Management

A system designer might want to write a DMA manager facility which accepts DMA requests from other software. The DMA manager software does not know in advance when new work requests are received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's NDPH and NDPL members point to the next descriptor, and the last descriptor points to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (Ix, Lx, Mx, and Bx registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the NDPH and NDPL members of each descriptor could even be written once at startup, and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

- Interrupt on every descriptor
- Interrupt minimally only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event is serviced separately (no interrupt overrun).

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

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a DMAx_CONFIG value with a FLOW value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts equal), the software increments its count and then starts the DMA unit by writing the new descriptor's DMAx_CONFIG value to the DMA channel's DMAx_CONFIG register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's DMAx_CONFIG value so that its upper half (FLOW and NDSIZE) now describes the newly-queued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMAx_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's DMAx_IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was queued too late (that is, the modification of the next-to-last descriptor's DMAx_CONFIG element occurred after that element was read into the DMA unit.) In this case, the interrupt handler writes the DMAx_CONFIG value appropriate for the last descriptor to the DMA channel's DMAx_CONFIG register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts needs to be complex, performing several MMR accesses to ensure robust operation. In such a system environment, a minimal interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's `DMAx_CONFIG` word should have a `FLOW` value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code queues later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the queue being processed by the DMA unit. In other words, all but the last active descriptors contain `FLOW` values >= 4 and have no interrupt enable set, while the last active descriptor contains a `FLOW` of 0 and an interrupt enable bit `DI_EN` set to 1. Also, all but the last waiting descriptors contain `FLOW` values >= 4 and no interrupt enables set, while the last waiting descriptor contains a `FLOW` of 0 and an interrupt enable bit `DI_EN` set to 1. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler by a single `DMAx_CONFIG` register write.

After queuing a new waiting descriptor, the non-interrupt software leaves a message for its interrupt handler in a memory mailbox location containing the desired DMAx_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting).

It is critical that the software not modify the contents of the active descriptor queue directly, once processing by the DMA unit is started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software never modifies descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel's DMAx_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler then passes a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMAx_CONFIG value of zero, indicating there is no more work to perform, then it passes an appropriate message (for example, zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler can 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 is 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 is 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 is stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the next descriptor pointer, regardless of whether operating in descriptor array mode or descriptor list mode.

The next descriptor pointer remains valid, if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the configuration register immediately.

One possible procedure is:

1. Write to DMAx_NEXT_DESC_PTR register.
2. Write to DMAx_CONFIG register with

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

3. Automatically fetched DMACFG register has
   FLOW = 0x0
   NDSIZE = 0x0
   SYNC = 1 (for transmitting DMAs only)
   DI_EN = 1
   DMAEN = 1.

4. In the interrupt routine, repeat step 2. The DMAx_NEXT_DESC_PTR register is updated by the descriptor fetch.

To avoid polling of the DMA_RUN bit, set the SYNC bit in case of memory read DMAs (DMA transmit or MDMA source).

If all DMACFG fields in a descriptor chain have the FLOW and NDSIZE fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

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 7-45). 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 7-7 on page 7-129. MDMA channels can be paused by software at any time by writing a 0 to the DRQ bit field in the HMDMAx_CONTROL register. This simply disables the self-generated DMA requests, regardless whether HMDMA is enabled or not.

DMA Registers

This section describes three categories of DMA registers:

- "DMA Channel Registers" on page 7-73
- "Handshake MDMA (HMDMA) Registers" on page 7-111
- "DMA Traffic Control Registers" on page 7-118

DMA Channel Registers

The processor features 24 peripheral DMA channels and four channel pairs for memory DMA. All channels have an identical set of registers summarized in Table 7-5 on page 7-74.

Table 7-5 on page 7-74 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, and the register category.

Table 7-5. Generic Names of the DMA Memory-Mapped Registers

MMR Offset	MMR Name	MMR Description
0x00	DMAx_NEXT_DESC_PTR MDMA_yy_NEXT_DESC_PTR	"Next Descriptor Pointer (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR) Registers" on page 7-106
0x04	DMAx_START_ADDR MDMA_yy_START_ADDR	"Start Address (DMAx_START_ADDR and MDMA_yy_START_ADDR) Registers" on page 7-88
0x08	DMAx_CONFIG MDMA_yy_CONFIG	"DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers" on page 7-79
0x0C	Reserved	Reserved
0x10	DMAx_X_COUNT MDMA_yy_X_COUNT	"Inner Loop Count (DMAx_X_COUNT and MDMA_yy_X_COUNT) Registers" on page 7-92
0x14	DMAx_X_MODIFY MDMA_yy_X_MODIFY	"Inner Loop Address Increment (DMAx_X_MODIFY and MDMA_yy_X_MODIFY) Registers" on page 7-97
0x18	DMAx_Y_COUNT MDMA_yy_Y_COUNT	"Outer Loop Count (DMAx_Y_COUNT and MDMA_yy_Y_COUNT) Registers" on page 7-99
0x1C	DMAx_Y_MODIFY MDMA_yy_Y_MODIFY	"Outer Loop Address Increment (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY) Registers" on page 7-103
0x20	DMAx_CURR_DESC_PTR MDMA_yy_CURR_DESC_PTR	"Current Descriptor Pointer (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR) Registers" on page 7-108
0x24	DMAx_CURR_ADDR MDMA_yy_CURR_ADDR	"Current Address (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR) Registers" on page 7-90
0x28	DMAx_IRQ_STATUS MDMA_yy_IRQ_STATUS	"Interrupt Status (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS) Registers" on page 7-84
0x2C	DMAx_PERIPHERAL_MAP MDMA_yy_PERIPHERAL_MAP	"Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers" on page 7-77

Table 7-5. Generic Names of the DMA Memory-Mapped Registers (Cont'd)

MMR Offset	MMR Name	MMR Description
0x30	DMAx_CURR_X_COUNT MDMA_yy_CURR_X_COUNT	"Current Inner Loop Count (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT) Registers" on page 7-94
0x34	Reserved	Reserved
0x38	DMAx_CURR_Y_COUNT MDMA_yy_CURR_Y_COUNT	"Current Outer Loop Count (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT) Registers" on page 7-101
0x3C	Reserved	Reserved

Channel-specific register names are shown in Table 7-5. For peripheral DMA channels, the prefix "DMAx_" is used where "x" stands for a channel number between 0 and 23. For memory DMA channels, the prefix is "MDMA_yy_", where "yy" stands for "D0", "D1", "D2", "D3", "S0", "S1", "S2" or "S3", and indicates the destination and source channel registers of MDMA0 through MDMA3. For example, the configuration register of peripheral DMA channel 6 is called DMA6_CONFIG, and the register for MDMA1 source channel is called MDMA_S1_CONFIG.

The generic MMR names shown in Table 7-5 on page 7-74 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 7-5 on page 7-74.
- Current registers, such as DMAx_CURR_ADDR and DMAx_CURR_X_COUNT
- Control/status registers, such as DMAx_IRQ_STATUS and DMAx_PERIPHERAL_MAP

All DMA registers can be accessed as 16-bit entities. The following registers, however, may also be accessed as 32-bit registers:

DMAx_NEXT_DESC_PTR

DMAx_START_ADDR

DMAx_CURR_DESC_PTR

DMAx_CURR_ADDR

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses different naming conventions for physical registers and their corresponding elements in descriptors that reside in memory. Table 7-5 on page 7-74 shows the relation.

Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers

Each DMA channel's peripheral map registers and addresses (`DMAx_PERIPHERAL_MAP` and `MDMA_yy_PERIPHERAL_MAP`, shown in Figure 7-7 and Table 7-6) contain bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Ensure that DMA is disabled on channels 6 and 7.
2. Write `DMA6_PERIPHERAL_MAP` with `0x7000` and `DMA7_PERIPHERAL_MAP` with `0x6000`.
3. Enable DMA on channels 6 and/or 7.

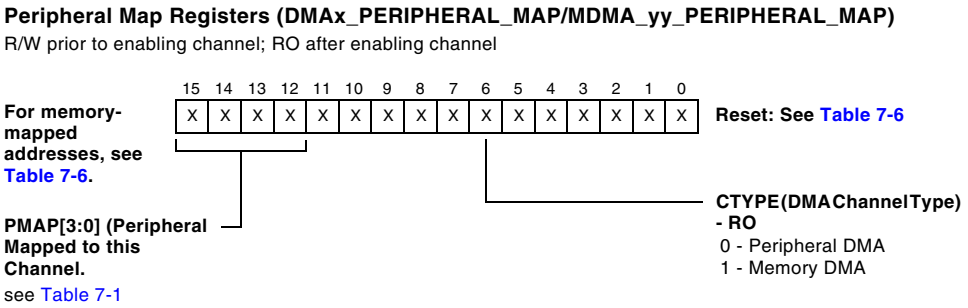


Figure 7-7. Peripheral Map Registers

Table 7-6. Peripheral Map Register Addresses and Reset Values

Register Name	Memory-Mapped Address	Reset Value
DMA0_PERIPHERAL_MAP	0xFFC0 0C2C	0x0000 (SPORT0 Rx)
DMA1_PERIPHERAL_MAP	0xFFC0 0C6C	0x1000
DMA2_PERIPHERAL_MAP	0xFFC0 0CAC	0x2000
DMA3_PERIPHERAL_MAP	0xFFC0 0CEC	0x3000
DMA4_PERIPHERAL_MAP	0xFFC0 0D2C	0x4000
DMA5_PERIPHERAL_MAP	0xFFC0 0D6C	0x5000
DMA6_PERIPHERAL_MAP	0xFFC0 0DAC	0x6000
DMA7_PERIPHERAL_MAP	0xFFC0 0DEC	0x7000
DMA8_PERIPHERAL_MAP	0xFFC0 0E2C	0x8000
DMA9_PERIPHERAL_MAP	0xFFC0 0E6C	0x9000
DMA10_PERIPHERAL_MAP	0xFFC0 0EAC	0xA000
DMA11_PERIPHERAL_MAP	0xFFC0 0EEC	0xB000
DMA12_PERIPHERAL_MAP	0xFFC0 1C2C	0x0000
DMA13_PERIPHERAL_MAP	0xFFC0 1C6C	0x1000
DMA14_PERIPHERAL_MAP	0xFFC0 1CAC	0x2000
DMA15_PERIPHERAL_MAP	0xFFC0 1CEC	0x3000
DMA16_PERIPHERAL_MAP	0xFFC0 1D2C	0x4000
DMA17_PERIPHERAL_MAP	0xFFC0 1D6C	0x5000
DMA18_PERIPHERAL_MAP	0xFFC0 1DAC	0x6000
DMA19_PERIPHERAL_MAP	0xFFC0 1DEC	0x7000
DMA20_PERIPHERAL_MAP	0xFFC0 1E2C	0x8000
DMA21_PERIPHERAL_MAP	0xFFC0 1E6C	0x9000
DMA22_PERIPHERAL_MAP	0xFFC0 1EAC	0xA000
DMA23_PERIPHERAL_MAP	0xFFC0 1EEC	0xB000
MDMA_D0_PERIPHERAL_MAP	0xFFC0 0F2C	0x0040

Table 7-6. Peripheral Map Register Addresses and Reset Values (Cont'd)

Register Name	Memory-Mapped Address	Reset Value
MDMA_S0_PERIPHERAL_MAP	0xFFC0 0F6C	0x0040
MDMA_D1_PERIPHERAL_MAP	0xFFC0 0FAC	0x0040
MDMA_S1_PERIPHERAL_MAP	0xFFC0 0FEC	0x0040
MDMA_D2_PERIPHERAL_MAP	0xFFC0 1F2C	0x0040
MDMA_S2_PERIPHERAL_MAP	0xFFC0 1F6C	0x0040
MDMA_D3_PERIPHERAL_MAP	0xFFC0 1FAC	0x0040
MDMA_S3_PERIPHERAL_MAP	0xFFC0 1FEC	0x0040

Table 7-1 on page 7-10 lists the peripheral map settings for each DMA-capable peripheral.

DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers

The DMA configuration registers and addresses (`DMAx_CONFIG` and `MDMA_yy_CONFIG`), shown in Figure 7-8 and Table 7-7, set up DMA parameters and operating modes. Note that writing the `DMAx_CONFIG` register while DMA is already running causes a DMA error unless writing with the `DMAEN` bit set to 0.

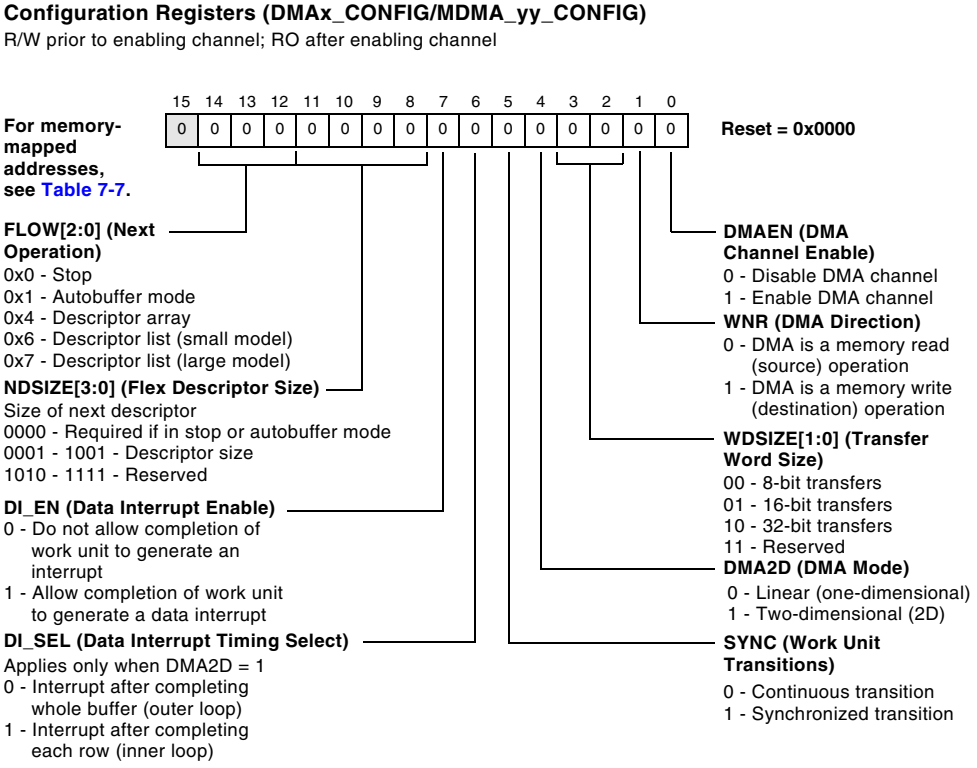


Figure 7-8. DMA Configuration Registers

Table 7-7. DMA Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CONFIG	0xFFC0 0C08
DMA1_CONFIG	0xFFC0 0C48
DMA2_CONFIG	0xFFC0 0C88
DMA3_CONFIG	0xFFC0 0CC8
DMA4_CONFIG	0xFFC0 0D08
DMA5_CONFIG	0xFFC0 0D48
DMA6_CONFIG	0xFFC0 0D88

Table 7-7. DMA Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA7_CONFIG	0xFFC0 0DC8
DMA8_CONFIG	0xFFC0 0E08
DMA9_CONFIG	0xFFC0 0E48
DMA10_CONFIG	0xFFC0 0E88
DMA11_CONFIG	0xFFC0 0EC8
DMA12_CONFIG	0xFFC0 1C08
DMA13_CONFIG	0xFFC0 1C48
DMA14_CONFIG	0xFFC0 1C88
DMA15_CONFIG	0xFFC0 1CC8
DMA16_CONFIG	0xFFC0 1D08
DMA17_CONFIG	0xFFC0 1D48
DMA18_CONFIG	0xFFC0 1D88
DMA19_CONFIG	0xFFC0 1DC8
DMA20_CONFIG	0xFFC0 1E08
DMA21_CONFIG	0xFFC0 1E48
DMA22_CONFIG	0xFFC0 1E88
DMA23_CONFIG	0xFFC0 1EC8
MDMA_D0_CONFIG	0xFFC0 0F08
MDMA_S0_CONFIG	0xFFC0 0F48
MDMA_D1_CONFIG	0xFFC0 0F88
MDMA_S1_CONFIG	0xFFC0 0FC8
MDMA_D2_CONFIG	0xFFC0 1F08
MDMA_S2_CONFIG	0xFFC0 1F48
MDMA_D3_CONFIG	0xFFC0 1F88
MDMA_S3_CONFIG	0xFFC0 1FC8

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

- FLOW[2:0] (next operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:
- 0x0 - stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The DMA_RUN status bit in the DMAx_IRQ_STATUS register changes from 1 to 0, while the DMAEN bit in the DMAx_CONFIG register is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to the DMAx_CONFIG register specifying the next work unit, in which the DMAEN bit is set to 1.

  0x1 - autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular buffer fashion based on user-programmed DMAx MMR settings. Upon completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMAEN bit in the 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 7-32.

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

Work unit transitions for MDMA streams are controlled by SYNC bit of the MDMA source channel's 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.

Interrupt Status (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS) Registers

The interrupt status registers and addresses (and MDMA_yy_IRQ_STATUS), shown in Figure 7-9 and Table 7-8, contain bits that record whether the DMA channel:

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

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

Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

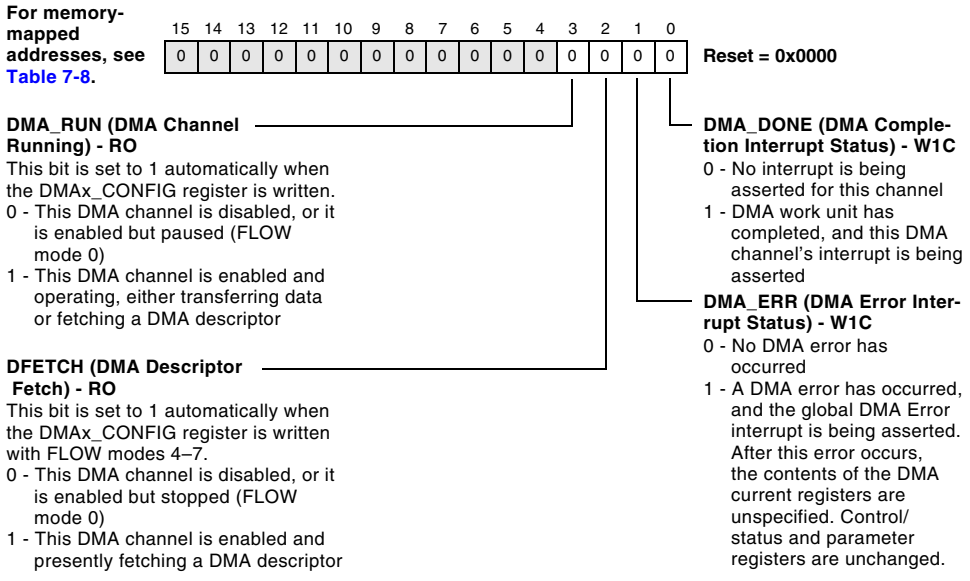


Figure 7-9. Interrupt Status Registers

For a memory transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event is signaled. It does not indicate the status of the DMA FIFO.

For MDMA transfers where it is not desired to use an interrupt to notify when the DMA operation has ended, software should poll the DMA_DONE bit, and not the DMA_RUN bit, to determine when the transaction has completed.

Table 7-8. Interrupt Status Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_IRQ_STATUS	0xFFC0 0C28
DMA1_IRQ_STATUS	0xFFC0 0C68
DMA2_IRQ_STATUS	0xFFC0 0CA8
DMA3_IRQ_STATUS	0xFFC0 0CE8
DMA4_IRQ_STATUS	0xFFC0 0D28
DMA5_IRQ_STATUS	0xFFC0 0D68
DMA6_IRQ_STATUS	0xFFC0 0DA8
DMA7_IRQ_STATUS	0xFFC0 0DE8
DMA8_IRQ_STATUS	0xFFC0 0E28
DMA9_IRQ_STATUS	0xFFC0 0E68
DMA10_IRQ_STATUS	0xFFC0 0EA8
DMA11_IRQ_STATUS	0xFFC0 0EE8
DMA12_IRQ_STATUS	0xFFC0 1C28
DMA13_IRQ_STATUS	0xFFC0 1C68
DMA14_IRQ_STATUS	0xFFC0 1CA8
DMA15_IRQ_STATUS	0xFFC0 1CE8
DMA16_IRQ_STATUS	0xFFC0 1D28
DMA17_IRQ_STATUS	0xFFC0 1D68
DMA18_IRQ_STATUS	0xFFC0 1DA8
DMA19_IRQ_STATUS	0xFFC0 1DE8
DMA20_IRQ_STATUS	0xFFC0 1E28

Table 7-8. Interrupt Status Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA21_IRQ_STATUS	0xFFC0 1E68
DMA22_IRQ_STATUS	0xFFC0 1EA8
DMA23_IRQ_STATUS	0xFFC0 1EE8
MDMA_D0_IRQ_STATUS	0xFFC0 0F28
MDMA_S0_IRQ_STATUS	0xFFC0 0F68
MDMA_D1_IRQ_STATUS	0xFFC0 0FA8
MDMA_S1_IRQ_STATUS	0xFFC0 0FE8
MDMA_D2_IRQ_STATUS	0xFFC0 1F28
MDMA_S2_IRQ_STATUS	0xFFC0 1F68
MDMA_D3_IRQ_STATUS	0xFFC0 1FA8
MDMA_S3_IRQ_STATUS	0xFFC0 1FE8

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

- Data driven interrupts (see Table 7-9)
- Peripheral error interrupts
- DMA error interrupts (for example, bad descriptor or bus error)

Separate interrupt request (IRQ) levels are allocated for data and peripheral error interrupts, and DMA error interrupts.

Table 7-9. Data Driven Interrupts

Interrupt Name	Description
No interrupt	Interrupts can be disabled for a given work unit.
Peripheral interrupt	These are peripheral (non-DMA) interrupts.

Table 7-9. Data Driven Interrupts (Cont'd)

Interrupt Name	Description
Row completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

The DMA error conditions for all DMA channels are OR'ed together into one system-level DMA error interrupt. The individual IRQ_STATUS words of each channel can be read to identify the channel that caused the DMA error interrupt.

The DMA_DONE and DMA_ERR interrupt indicators are write-1-to-clear (W1C).

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

Start Address (DMAx_START_ADDR and MDMA_yy_START_ADDR) Registers

The start address registers and addresses (DMAx_START_ADDR and MDMA_yy_START_ADDR), shown in Figure 7-10 and Table 7-10, contain the start address of the data buffer currently targeted for DMA.

Start Address Registers (DMAx_START_ADDR/ MDMA_yy_START_ADDR)

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

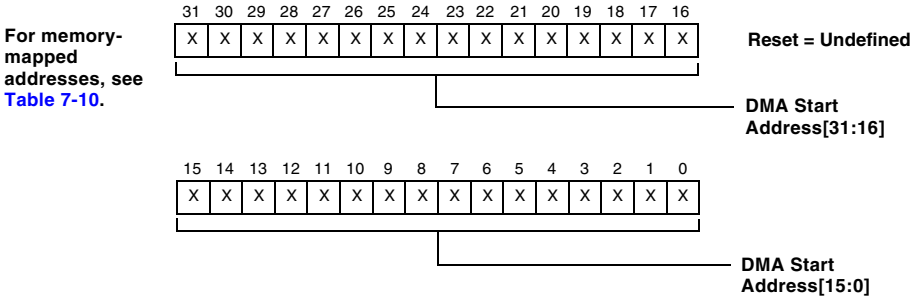


Figure 7-10. Start Address Registers

Table 7-10. Start Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_START_ADDR	0xFFC0 0C04
DMA1_START_ADDR	0xFFC0 0C44
DMA2_START_ADDR	0xFFC0 0C84
DMA3_START_ADDR	0xFFC0 0CC4
DMA4_START_ADDR	0xFFC0 0D04
DMA5_START_ADDR	0xFFC0 0D44
DMA6_START_ADDR	0xFFC0 0D84
DMA7_START_ADDR	0xFFC0 0DC4
DMA8_START_ADDR	0xFFC0 0E04
DMA9_START_ADDR	0xFFC0 0E44
DMA10_START_ADDR	0xFFC0 0E84
DMA11_START_ADDR	0xFFC0 0EC4
DMA12_START_ADDR	0xFFC0 1C04
DMA13_START_ADDR	0xFFC0 1C44
DMA14_START_ADDR	0xFFC0 1C84

Table 7-10. Start Address Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA15_START_ADDR	0xFFC0 1CC4
DMA16_START_ADDR	0xFFC0 1D04
DMA17_START_ADDR	0xFFC0 1D44
DMA18_START_ADDR	0xFFC0 1D84
DMA19_START_ADDR	0xFFC0 1DC4
DMA20_START_ADDR	0xFFC0 1E04
DMA21_START_ADDR	0xFFC0 1E44
DMA22_START_ADDR	0xFFC0 1E84
DMA23_START_ADDR	0xFFC0 1EC4
MDMA_D0_START_ADDR	0xFFC0 0F04
MDMA_S0_START_ADDR	0xFFC0 0F44
MDMA_D1_START_ADDR	0xFFC0 0F84
MDMA_S1_START_ADDR	0xFFC0 0FC4
MDMA_D2_START_ADDR	0xFFC0 1F04
MDMA_S2_START_ADDR	0xFFC0 1F44
MDMA_D3_START_ADDR	0xFFC0 1F84
MDMA_S3_START_ADDR	0xFFC0 1FC4

Current Address (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR) Registers

The current address registers and addresses (`DMAx_CURR_ADDR` and `MDMA_yy_CURR_ADDR`), shown in Figure 7-11 and Table 7-11, contain 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.

Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)
R/W prior to enabling channel; RO after enabling channel

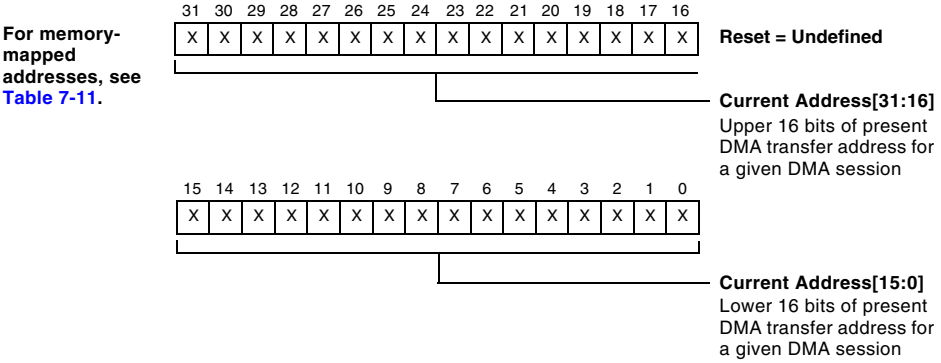


Figure 7-11. Current Address Registers

Table 7-11. Current Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_ADDR	0xFFC0 0C24
DMA1_CURR_ADDR	0xFFC0 0C64
DMA2_CURR_ADDR	0xFFC0 0CA4
DMA3_CURR_ADDR	0xFFC0 0CE4
DMA4_CURR_ADDR	0xFFC0 0D24
DMA5_CURR_ADDR	0xFFC0 0D64
DMA6_CURR_ADDR	0xFFC0 0DA4
DMA7_CURR_ADDR	0xFFC0 0DE4
DMA8_CURR_ADDR	0xFFC0 0E24
DMA9_CURR_ADDR	0xFFC0 0E64
DMA10_CURR_ADDR	0xFFC0 0EA4
DMA11_CURR_ADDR	0xFFC0 0EE4
DMA12_CURR_ADDR	0xFFC0 1C24
DMA13_CURR_ADDR	0xFFC0 1C64

Table 7-11. Current Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA14_CURR_ADDR	0xFFC0 1CA4
DMA15_CURR_ADDR	0xFFC0 1CE4
DMA16_CURR_ADDR	0xFFC0 1D24
DMA17_CURR_ADDR	0xFFC0 1D64
DMA18_CURR_ADDR	0xFFC0 1DA4
DMA19_CURR_ADDR	0xFFC0 1DE4
DMA20_CURR_ADDR	0xFFC0 1E24
DMA21_CURR_ADDR	0xFFC0 1E64
DMA22_CURR_ADDR	0xFFC0 1EA4
DMA23_CURR_ADDR	0xFFC0 1EE4
MDMA_D0_CURR_ADDR	0xFFC0 0F24
MDMA_S0_CURR_ADDR	0xFFC0 0F64
MDMA_D1_CURR_ADDR	0xFFC0 0FA4
MDMA_S1_CURR_ADDR	0xFFC0 0FE4
MDMA_D2_CURR_ADDR	0xFFC0 1F24
MDMA_S2_CURR_ADDR	0xFFC0 1F64
MDMA_D3_CURR_ADDR	0xFFC0 1FA4
MDMA_S3_CURR_ADDR	0xFFC0 1FE4

Inner Loop Count (DMAx_X_COUNT and MDMA_yy_X_COUNT) Registers

For 2D DMA, the inner loop count registers and addresses (`DMAx_X_COUNT` and `MDMA_yy_X_COUNT`), shown in Figure 7-12 and Table 7-12, contain the inner loop count. For 1D DMA, it specifies the number of elements to read in. For details, see "Two-Dimensional DMA Operation" on page 7-19. A value of 0 in `DMAx_X_COUNT` corresponds to 65,536 elements.

Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)
R/W prior to enabling channel; RO after enabling channel

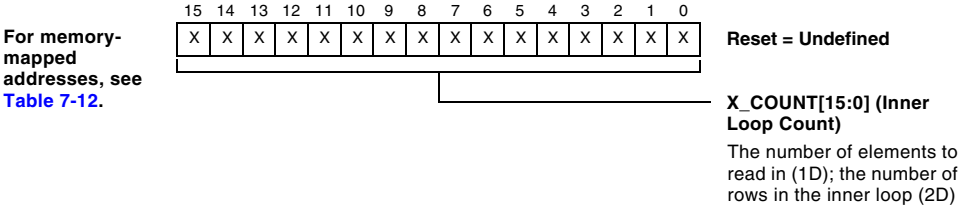


Figure 7-12. Inner Loop Count Registers

Table 7-12. Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_COUNT	0xFFC0 0C10
DMA1_X_COUNT	0xFFC0 0C50
DMA2_X_COUNT	0xFFC0 0C90
DMA3_X_COUNT	0xFFC0 0CD0
DMA4_X_COUNT	0xFFC0 0D10
DMA5_X_COUNT	0xFFC0 0D50
DMA6_X_COUNT	0xFFC0 0D90
DMA7_X_COUNT	0xFFC0 0DD0
DMA8_X_COUNT	0xFFC0 0E10
DMA9_X_COUNT	0xFFC0 0E50
DMA10_X_COUNT	0xFFC0 0E90
DMA11_X_COUNT	0xFFC0 0ED0
DMA12_X_COUNT	0xFFC0 1C10
DMA13_X_COUNT	0xFFC0 1C50
DMA14_X_COUNT	0xFFC0 1C90
DMA15_X_COUNT	0xFFC0 1CD0
DMA16_X_COUNT	0xFFC0 1D10

Table 7-12. Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA17_X_COUNT	0xFFC0 1D50
DMA18_X_COUNT	0xFFC0 1D90
DMA19_X_COUNT	0xFFC0 1DD0
DMA20_X_COUNT	0xFFC0 1E10
DMA21_X_COUNT	0xFFC0 1E50
DMA22_X_COUNT	0xFFC0 1E90
DMA23_X_COUNT	0xFFC0 1ED0
MDMA_D0_X_COUNT	0xFFC0 0F10
MDMA_S0_X_COUNT	0xFFC0 0F50
MDMA_D1_X_COUNT	0xFFC0 0F90
MDMA_S1_X_COUNT	0xFFC0 0FD0
MDMA_D2_X_COUNT	0xFFC0 1F10
MDMA_S2_X_COUNT	0xFFC0 1F50
MDMA_D3_X_COUNT	0xFFC0 1F90
MDMA_S3_X_COUNT	0xFFC0 1FD0

Current Inner Loop Count (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT) Registers

The current inner loop count registers and addresses (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT), shown in Figure 7-13 and Table 7-13, hold the number of transfers remaining in the current DMA row (inner loop).

On the first memory transfer of each DMA work unit, it is loaded with the value in the DMAx_X_COUNT register and then decremented. For 2D DMA, on the last memory transfer in each row except the last row, it is reloaded with the value in the DMAx_X_COUNT register; this occurs at the same time that the value in the DMAx_CURR_Y_COUNT register is

decremented. Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete.

In 2D DMA, the DMAx_CURR_X_COUNT register value is 0 only when the entire transfer is complete. Between rows it is equal to the value of the DMAx_X_COUNT register.

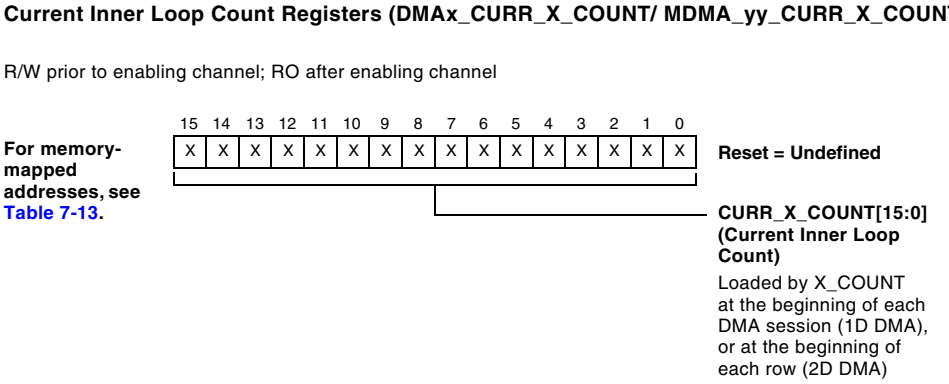


Figure 7-13. Current Inner Loop Count Registers

Table 7-13. Current Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_X_COUNT	0xFFC0 0C30
DMA1_CURR_X_COUNT	0xFFC0 0C70
DMA2_CURR_X_COUNT	0xFFC0 0CB0
DMA3_CURR_X_COUNT	0xFFC0 0CF0
DMA4_CURR_X_COUNT	0xFFC0 0D30
DMA5_CURR_X_COUNT	0xFFC0 0D70
DMA6_CURR_X_COUNT	0xFFC0 0DB0
DMA7_CURR_X_COUNT	0xFFC0 0DF0

Table 7-13. Current Inner Loop Count Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA8_CURR_X_COUNT	0xFFC0 0E30
DMA9_CURR_X_COUNT	0xFFC0 0E70
DMA10_CURR_X_COUNT	0xFFC0 0EB0
DMA11_CURR_X_COUNT	0xFFC0 0EF0
DMA12_CURR_X_COUNT	0xFFC0 1C30
DMA13_CURR_X_COUNT	0xFFC0 1C70
DMA14_CURR_X_COUNT	0xFFC0 1CB0
DMA15_CURR_X_COUNT	0xFFC0 1CF0
DMA16_CURR_X_COUNT	0xFFC0 1D30
DMA17_CURR_X_COUNT	0xFFC0 1D70
DMA18_CURR_X_COUNT	0xFFC0 1DB0
DMA19_CURR_X_COUNT	0xFFC0 1DF0
DMA20_CURR_X_COUNT	0xFFC0 1E30
DMA21_CURR_X_COUNT	0xFFC0 1E70
DMA22_CURR_X_COUNT	0xFFC0 1EB0
DMA23_CURR_X_COUNT	0xFFC0 1EF0
MDMA_D0_CURR_X_COUNT	0xFFC0 0F30
MDMA_S0_CURR_X_COUNT	0xFFC0 0F70
MDMA_D1_CURR_X_COUNT	0xFFC0 0FB0
MDMA_S1_CURR_X_COUNT	0xFFC0 0FF0
MDMA_D2_CURR_X_COUNT	0xFFC0 1F30
MDMA_S2_CURR_X_COUNT	0xFFC0 1F70
MDMA_D3_CURR_X_COUNT	0xFFC0 1FB0
MDMA_S3_CURR_X_COUNT	0xFFC0 1FF0

Inner Loop Address Increment (DMAx_X_MODIFY and MDMA_yy_X_MODIFY) Registers

The inner loop address increment registers and addresses (DMAx_X_MODIFY and MDMA_yy_X_MODIFY), shown in Figure 7-14 and Table 7-14, contain a signed, two's-complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

DMAx_X_MODIFY is specified in bytes, regardless of the DMA transfer size.

In 2D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, the DMAx_Y_MODIFY register is applied instead, except on the very last transfer of each work unit. The DMAx_X_MODIFY register is always applied on the last transfer of a work unit.

The DMAx_X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

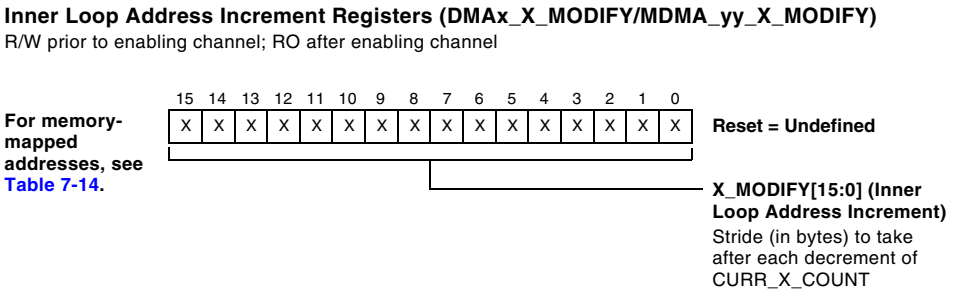


Figure 7-14. Inner Loop Address Increment Registers

Table 7-14. Inner Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_MODIFY	0xFFC0 0C14
DMA1_X_MODIFY	0xFFC0 0C54
DMA2_X_MODIFY	0xFFC0 0C94
DMA3_X_MODIFY	0xFFC0 0CD4
DMA4_X_MODIFY	0xFFC0 0D14
DMA5_X_MODIFY	0xFFC0 0D54
DMA6_X_MODIFY	0xFFC0 0D94
DMA7_X_MODIFY	0xFFC0 0DD4
DMA8_X_MODIFY	0xFFC0 0E14
DMA9_X_MODIFY	0xFFC0 0E54
DMA10_X_MODIFY	0xFFC0 0E94
DMA11_X_MODIFY	0xFFC0 0ED4
DMA12_X_MODIFY	0xFFC0 1C14
DMA13_X_MODIFY	0xFFC0 1C54
DMA14_X_MODIFY	0xFFC0 1C94
DMA15_X_MODIFY	0xFFC0 1CD4
DMA16_X_MODIFY	0xFFC0 1D14
DMA17_X_MODIFY	0xFFC0 1D54
DMA18_X_MODIFY	0xFFC0 1D94
DMA19_X_MODIFY	0xFFC0 1DD4
DMA20_X_MODIFY	0xFFC0 1E14
DMA21_X_MODIFY	0xFFC0 1E54
DMA22_X_MODIFY	0xFFC0 1E94
DMA23_X_MODIFY	0xFFC0 1ED4
MDMA_D0_X_MODIFY	0xFFC0 0F14

Table 7-14. Inner Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_S0_X_MODIFY	0xFFC0 0F54
MDMA_D1_X_MODIFY	0xFFC0 0F94
MDMA_S1_X_MODIFY	0xFFC0 0FD4
MDMA_D2_X_MODIFY	0xFFC0 1F14
MDMA_S2_X_MODIFY	0xFFC0 1F54
MDMA_D3_X_MODIFY	0xFFC0 1F94
MDMA_S3_X_MODIFY	0xFFC0 1FD4

Outer Loop Count (DMAx_Y_COUNT and MDMA_yy_Y_COUNT) Registers

For 2D DMA, the outer loop count registers and addresses (`DMAx_Y_COUNT` and `MDMA_yy_Y_COUNT`), shown in Figure 7-15 and Table 7-15, contain the outer loop count. It is not used in 1D DMA mode. This register contains the number of rows in the outer loop of a 2D DMA sequence. For details, see "Two-Dimensional DMA Operation" on page 7-19.

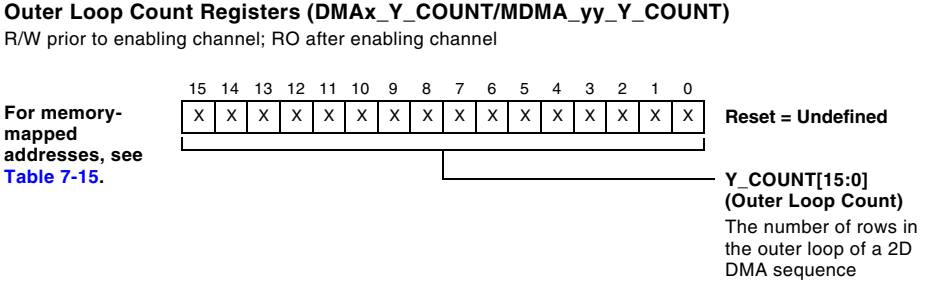


Figure 7-15. Outer Loop Count Registers

Table 7-15. Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_COUNT	0xFFC0 0C18
DMA1_Y_COUNT	0xFFC0 0C58
DMA2_Y_COUNT	0xFFC0 0C98
DMA3_Y_COUNT	0xFFC0 0CD8
DMA4_Y_COUNT	0xFFC0 0D18
DMA5_Y_COUNT	0xFFC0 0D58
DMA6_Y_COUNT	0xFFC0 0D98
DMA7_Y_COUNT	0xFFC0 0DD8
DMA8_Y_COUNT	0xFFC0 0E18
DMA9_Y_COUNT	0xFFC0 0E58
DMA10_Y_COUNT	0xFFC0 0E98
DMA11_Y_COUNT	0xFFC0 0ED8
DMA12_Y_COUNT	0xFFC0 1C18
DMA13_Y_COUNT	0xFFC0 1C58
DMA14_Y_COUNT	0xFFC0 1C98
DMA15_Y_COUNT	0xFFC0 1CD8
DMA16_Y_COUNT	0xFFC0 1D18
DMA17_Y_COUNT	0xFFC0 1D58
DMA18_Y_COUNT	0xFFC0 1D98
DMA19_Y_COUNT	0xFFC0 1DD8
DMA20_Y_COUNT	0xFFC0 1E18
DMA21_Y_COUNT	0xFFC0 1E58
DMA22_Y_COUNT	0xFFC0 1E98
DMA23_Y_COUNT	0xFFC0 1ED8
MDMA_D0_Y_COUNT	0xFFC0 0F18
MDMA_S0_Y_COUNT	0xFFC0 0F58

Table 7-15. Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
MDMA_D1_Y_COUNT	0xFFC0 0F98
MDMA_S1_Y_COUNT	0xFFC0 0FD8
MDMA_D2_Y_COUNT	0xFFC0 1F18
MDMA_S2_Y_COUNT	0xFFC0 1F58
MDMA_D3_Y_COUNT	0xFFC0 1F98
MDMA_S3_Y_COUNT	0xFFC0 1FD8

Current Outer Loop Count (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT) Registers

The current outer loop count registers and addresses (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT), shown in Figure 7-16 and Table 7-16, used only in 2D mode, hold the number of full or partial rows (outer loops) remaining in the current work unit.

On the first memory transfer of each DMA work unit, it is loaded with the value of the DMAx_Y_COUNT register. The register is decremented each time the DMAx_CURR_X_COUNT register expires during 2D DMA operation (1 to DMAx_X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2D DMA session is complete, DMAx_CURR_Y_COUNT = 1 and DMAx_CURR_X_COUNT = 0.

Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT)

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

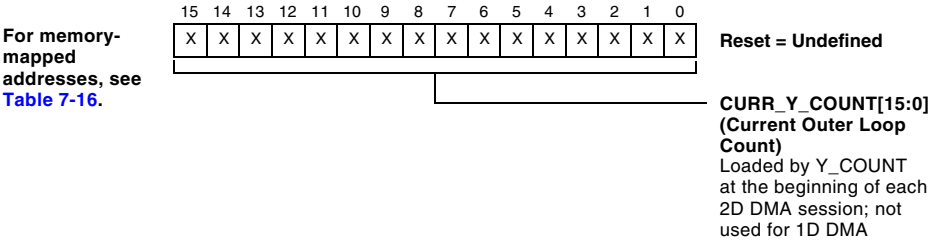


Figure 7-16. Current Outer Loop Count Registers

Table 7-16. Current Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_Y_COUNT	0xFFC0 0C38
DMA1_CURR_Y_COUNT	0xFFC0 0C78
DMA2_CURR_Y_COUNT	0xFFC0 0CB8
DMA3_CURR_Y_COUNT	0xFFC0 0CF8
DMA4_CURR_Y_COUNT	0xFFC0 0D38
DMA5_CURR_Y_COUNT	0xFFC0 0D78
DMA6_CURR_Y_COUNT	0xFFC0 0DB8
DMA7_CURR_Y_COUNT	0xFFC0 0DF8
DMA8_CURR_Y_COUNT	0xFFC0 0E38
DMA9_CURR_Y_COUNT	0xFFC0 0E78
DMA10_CURR_Y_COUNT	0xFFC0 0EB8
DMA11_CURR_Y_COUNT	0xFFC0 0EF8
DMA12_CURR_Y_COUNT	0xFFC0 1C38
DMA13_CURR_Y_COUNT	0xFFC0 1C78
DMA14_CURR_Y_COUNT	0xFFC0 1CB8

Table 7-16. Current Outer Loop Count Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA15_CURR_Y_COUNT	0xFFC0 1CF8
DMA16_CURR_Y_COUNT	0xFFC0 1D38
DMA17_CURR_Y_COUNT	0xFFC0 1D78
DMA18_CURR_Y_COUNT	0xFFC0 1DB8
DMA19_CURR_Y_COUNT	0xFFC0 1DF8
DMA20_CURR_Y_COUNT	0xFFC0 1E38
DMA21_CURR_Y_COUNT	0xFFC0 1E78
DMA22_CURR_Y_COUNT	0xFFC0 1EB8
DMA23_CURR_Y_COUNT	0xFFC0 1EF8
MDMA_D0_CURR_Y_COUNT	0xFFC0 0F38
MDMA_S0_CURR_Y_COUNT	0xFFC0 0F78
MDMA_D1_CURR_Y_COUNT	0xFFC0 0FB8
MDMA_S1_CURR_Y_COUNT	0xFFC0 0FF8
MDMA_D2_CURR_Y_COUNT	0xFFC0 1F38
MDMA_S2_CURR_Y_COUNT	0xFFC0 1F78
MDMA_D3_CURR_Y_COUNT	0xFFC0 1FB8
MDMA_S3_CURR_Y_COUNT	0xFFC0 1FF8

Outer Loop Address Increment (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY) Registers

The outer loop address increment registers and addresses (`DMAx_Y_MODIFY` and `MDMA_yy_Y_MODIFY`), shown in Figure 7-17 and Table 7-17, contain a signed, two's-complement value. This byte-address increment is applied after each decrement of the `DMAx_CURR_Y_COUNT` register except for the last item in the 2D array where the `DMAx_CURR_Y_COUNT` also expires. The value

is the offset between the last word of one “row” and the first word of the next “row.” For details, see “Two-Dimensional DMA Operation” on page 7-19.

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

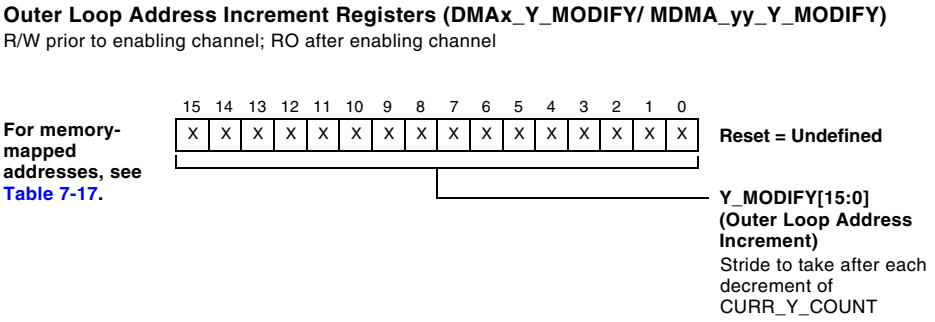


Figure 7-17. Outer Loop Address Increment Registers

Table 7-17. Outer Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_MODIFY	0xFFC0 0C1C
DMA1_Y_MODIFY	0xFFC0 0C5C
DMA2_Y_MODIFY	0xFFC0 0C9C
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

Table 7-17. Outer Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA10_Y_MODIFY	0xFFC0 0E9C
DMA11_Y_MODIFY	0xFFC0 0EDC
DMA12_Y_MODIFY	0xFFC0 1C1C
DMA13_Y_MODIFY	0xFFC0 1C5C
DMA14_Y_MODIFY	0xFFC0 1C9C
DMA15_Y_MODIFY	0xFFC0 1CDC
DMA16_Y_MODIFY	0xFFC0 1D1C
DMA17_Y_MODIFY	0xFFC0 1D5C
DMA18_Y_MODIFY	0xFFC0 1D9C
DMA19_Y_MODIFY	0xFFC0 1DDC
DMA20_Y_MODIFY	0xFFC0 1E1C
DMA21_Y_MODIFY	0xFFC0 1E5C
DMA22_Y_MODIFY	0xFFC0 1E9C
DMA23_Y_MODIFY	0xFFC0 1EDC
MDMA_D0_Y_MODIFY	0xFFC0 0F1C
MDMA_S0_Y_MODIFY	0xFFC0 0F5C
MDMA_D1_Y_MODIFY	0xFFC0 0F9C
MDMA_S1_Y_MODIFY	0xFFC0 0FDC
MDMA_D2_Y_MODIFY	0xFFC0 1F1C
MDMA_S2_Y_MODIFY	0xFFC0 1F5C
MDMA_D3_Y_MODIFY	0xFFC0 1F9C
MDMA_S3_Y_MODIFY	0xFFC0 1FDC

Next Descriptor Pointer (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR) Registers

The next descriptor pointer registers and addresses (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR), shown in Figure 7-18 and Table 7-18, specify where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, the 32-bit DMAx_NEXT_DESC_PTR register is copied into the DMAx_CURR_DESC_PTR register. Then, during the descriptor fetch, the DMAx_CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, the DMAx_NEXT_DESC_PTR register, and not the DMAx_CURR_DESC_PTR register, must be programmed directly through MMR access before starting DMA operation.

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

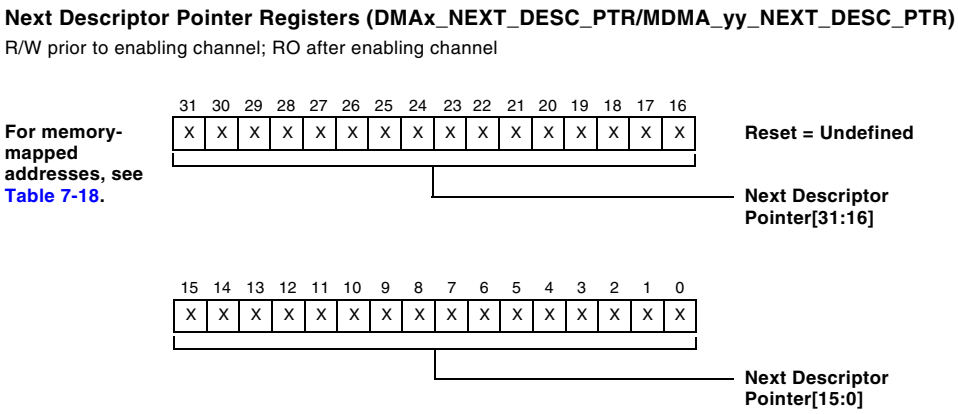


Figure 7-18. Next Descriptor Pointer Registers

Table 7-18. Next Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_NEXT_DESC_PTR	0xFFC0 0C00
DMA1_NEXT_DESC_PTR	0xFFC0 0C40
DMA2_NEXT_DESC_PTR	0xFFC0 0C80
DMA3_NEXT_DESC_PTR	0xFFC0 0CC0
DMA4_NEXT_DESC_PTR	0xFFC0 0D00
DMA5_NEXT_DESC_PTR	0xFFC0 0D40
DMA6_NEXT_DESC_PTR	0xFFC0 0D80
DMA7_NEXT_DESC_PTR	0xFFC0 0DC0
DMA8_NEXT_DESC_PTR	0xFFC0 0E00
DMA9_NEXT_DESC_PTR	0xFFC0 0E40
DMA10_NEXT_DESC_PTR	0xFFC0 0E80
DMA11_NEXT_DESC_PTR	0xFFC0 0EC0
DMA12_NEXT_DESC_PTR	0xFFC0 1C00
DMA13_NEXT_DESC_PTR	0xFFC0 1C40
DMA14_NEXT_DESC_PTR	0xFFC0 1C80
DMA15_NEXT_DESC_PTR	0xFFC0 1CC0
DMA16_NEXT_DESC_PTR	0xFFC0 1D00
DMA17_NEXT_DESC_PTR	0xFFC0 1D40
DMA18_NEXT_DESC_PTR	0xFFC0 1D80
DMA19_NEXT_DESC_PTR	0xFFC0 1DC0
DMA20_NEXT_DESC_PTR	0xFFC0 1E00
DMA21_NEXT_DESC_PTR	0xFFC0 1E40
DMA22_NEXT_DESC_PTR	0xFFC0 1E80
DMA23_NEXT_DESC_PTR	0xFFC0 1EC0
MDMA_D0_NEXT_DESC_PTR	0xFFC0 0F00

Table 7-18. Next Descriptor Pointer Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_S0_NEXT_DESC_PTR	0xFFC0 0F40
MDMA_D1_NEXT_DESC_PTR	0xFFC0 0F80
MDMA_S1_NEXT_DESC_PTR	0xFFC0 0FC0
MDMA_D2_NEXT_DESC_PTR	0xFFC0 1F00
MDMA_S2_NEXT_DESC_PTR	0xFFC0 1F40
MDMA_D3_NEXT_DESC_PTR	0xFFC0 1F80
MDMA_S3_NEXT_DESC_PTR	0xFFC0 1FC0

Current Descriptor Pointer (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR) Registers

The current descriptor pointer registers and addresses (`DMAx_CURR_DESC_PTR` and `MDMA_yy_CURR_DESC_PTR`), shown in Figure 7-19 and Table 7-19, contain the memory address for the next descriptor element to be loaded. For `FLOW` mode settings that involve descriptors (`FLOW` = 4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (`FLOW` = 6 or 7), this register is initialized from the `DMAx_NEXT_DESC_PTR` register before loading each descriptor. Then, the address in the `DMAx_CURR_DESC_PTR` register increments as each descriptor element is read in.

When the entire descriptor is read, the `DMAx_CURR_DESC_PTR` register contains this value:

```
Descriptor Start Address + (2 x Descriptor Size) (# of elements)
```

For descriptor array mode (`FLOW = 4`), this register, and not the `DMAx_NEXT_DESC_PTR` register, must be programmed by MMR access before starting DMA operation.

Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

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

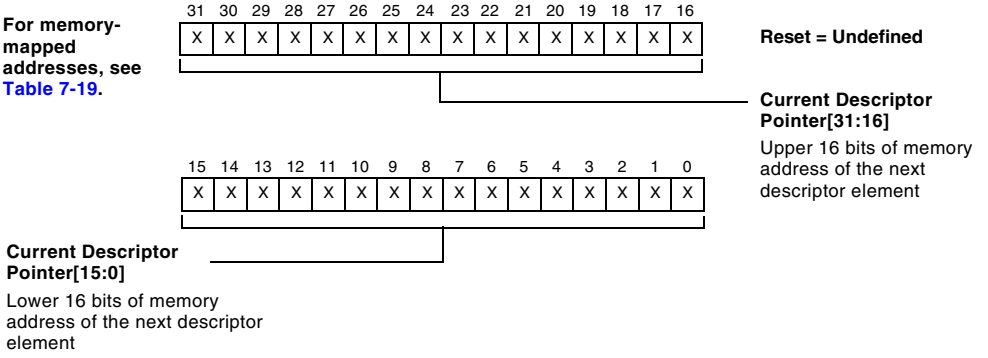


Figure 7-19. Current Descriptor Pointer Registers

Table 7-19. Current Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_DESC_PTR	0xFFC0 0C20
DMA1_CURR_DESC_PTR	0xFFC0 0C60
DMA2_CURR_DESC_PTR	0xFFC0 0CA0
DMA3_CURR_DESC_PTR	0xFFC0 0CE0
DMA4_CURR_DESC_PTR	0xFFC0 0D20
DMA5_CURR_DESC_PTR	0xFFC0 0D60
DMA6_CURR_DESC_PTR	0xFFC0 0DA0
DMA7_CURR_DESC_PTR	0xFFC0 0DE0
DMA8_CURR_DESC_PTR	0xFFC0 0E20
DMA9_CURR_DESC_PTR	0xFFC0 0E60
DMA10_CURR_DESC_PTR	0xFFC0 0EA0
DMA11_CURR_DESC_PTR	0xFFC0 0EE0

Table 7-19. Current Descriptor Pointer Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA12_CURR_DESC_PTR	0xFFC0 1C20
DMA13_CURR_DESC_PTR	0xFFC0 1C60
DMA14_CURR_DESC_PTR	0xFFC0 1CA0
DMA15_CURR_DESC_PTR	0xFFC0 1CE0
DMA16_CURR_DESC_PTR	0xFFC0 1D20
DMA17_CURR_DESC_PTR	0xFFC0 1D60
DMA18_CURR_DESC_PTR	0xFFC0 1DA0
DMA19_CURR_DESC_PTR	0xFFC0 1DE0
DMA20_CURR_DESC_PTR	0xFFC0 1E20
DMA21_CURR_DESC_PTR	0xFFC0 1E60
DMA22_CURR_DESC_PTR	0xFFC0 1EA0
DMA23_CURR_DESC_PTR	0xFFC0 1EE0
MDMA_D0_CURR_DESC_PTR	0xFFC0 0F20
MDMA_S0_CURR_DESC_PTR	0xFFC0 0F60
MDMA_D1_CURR_DESC_PTR	0xFFC0 0FA0
MDMA_S1_CURR_DESC_PTR	0xFFC0 0FE0
MDMA_D2_CURR_DESC_PTR	0xFFC0 1F20
MDMA_S2_CURR_DESC_PTR	0xFFC0 1F60
MDMA_D3_CURR_DESC_PTR	0xFFC0 1FA0
MDMA_S3_CURR_DESC_PTR	0xFFC0 1FE0

Handshake MDMA (HMDMA) Registers

The Blackfin processor features two HMDMA blocks. HMDMA0 is associated with MDMA2, and HMDMA1 is associated with MDMA3. Table 7-20 lists the naming conventions for these registers.

Table 7-20. Naming Conventions for Handshake MDMA Registers

Handshake MDMA MMR Name (x = 0 or 1)	Memory-Mapped Address
HMDMA0_CONTROL (on page 7-111)	0xFFC0 4500
HMDMA0_ECINIT (on page 7-117)	0xFFC0 4504
HMDMA0_BCINIT (on page 7-114)	0xFFC0 4508
HMDMA0_ECURGENT (on page 7-117)	0xFFC0 450C
HMDMA0_ECOVERFLOW (on page 7-118)	0xFFC0 4510
HMDMA0_ECOUNT (on page 7-116)	0xFFC0 4514
HMDMA0_BCOUNT (on page 7-115)	0xFFC0 4518
HMDMA1_CONTROL (on page 7-111)	0xFFC0 4540
HMDMA1_ECINIT (on page 7-117)	0xFFC0 4544
HMDMA1_BCINIT (on page 7-114)	0xFFC0 4548
HMDMA1_ECURGENT (on page 7-117)	0xFFC0 454C
HMDMA1_ECOVERFLOW (on page 7-118)	0xFFC0 4550
HMDMA1_ECOUNT (on page 7-116)	0xFFC0 4554
HMDMA1_BCOUNT (on page 7-115)	0xFFC0 4558

Handshake MDMA Control (HMDMAx_CONTROL) Registers

The handshake MDMA control registers (`HMDMAx_CONTROL`), shown in Figure 7-20, set up HMDMA parameters and operating modes.

The DRQ[1:0] field is used to control the priority of the MDMA channel when the HMDMA is disabled, that is, when handshake control is not being used (see Table 7-21).

Table 7-21. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
b#00	Disabled	The MDMA request is disabled.
b#01	Enabled/S	Normal MDMA channel priority. The channel in this mode is limited to single memory transfers separated by one idle system clock. Request single transfer from MDMA channel.
b#10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
b#11	Urgent	The MDMA channel priority is elevated to urgent. In this state, it has higher priority for memory access than non-urgent channels. If two channels are both urgent, the lower-numbered channel has priority.

The RBC bit forces the BCOUNT register to be reloaded with the BCINIT value while the module is already active. Do not set this bit in the same write that sets the HMDMAEN bit to active.

The HMDMA0_CONTROL[10:11] bits are used to control the gating of Core, DMAC0, PIXC and MDMA during EPPI urgency conditions. For more information see “Elevating EPPI Urgent Requests at DDR Controller Interface” on page 15-71.

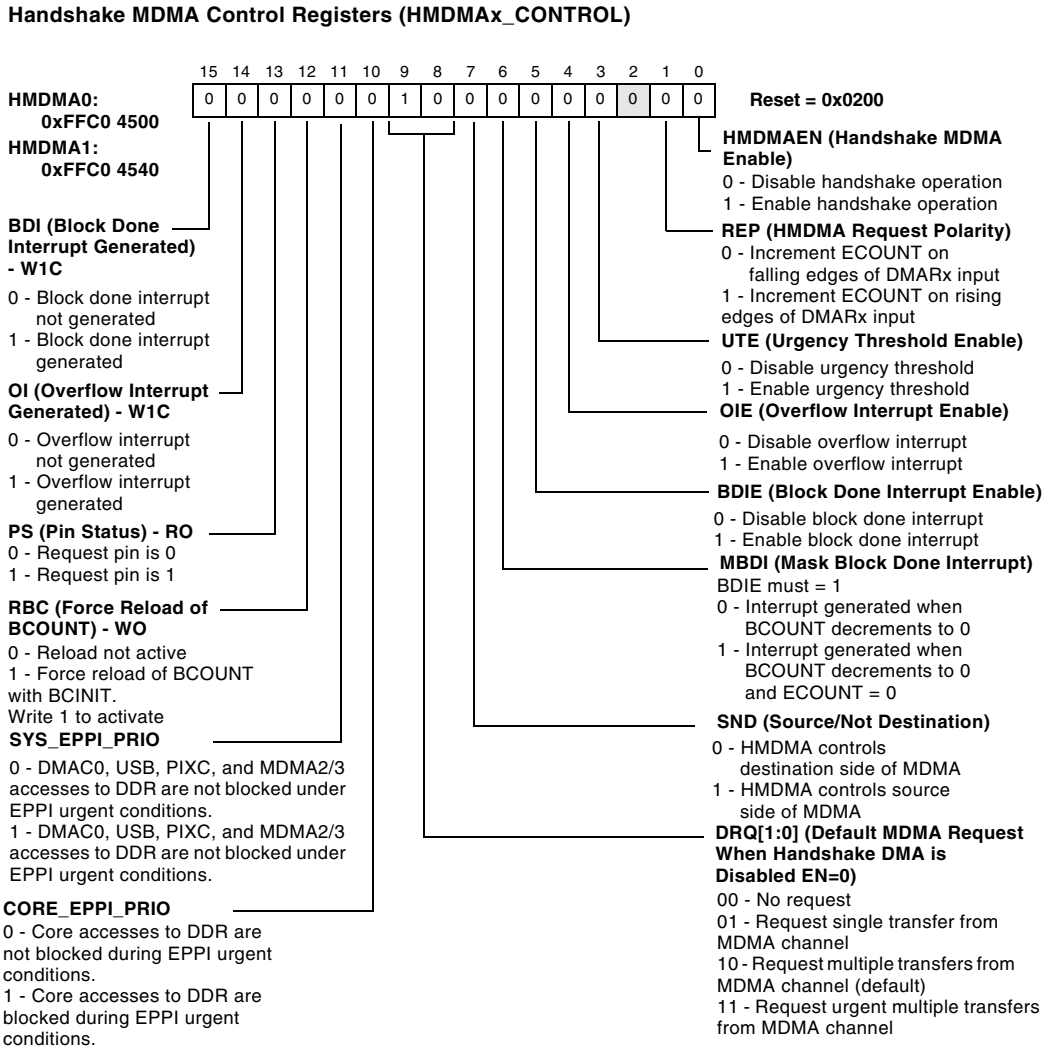


Figure 7-20. Handshake MDMA Control Registers

Table 7-22. EPPI_DMA_URGENT_ACCESS

HMDMA0[11]	HMDMA0[10]	Action
0	0	Do not gate-off core, PIXC or DMAC0 on EPPI(0,1 2) urgent conditions
0	1	Gate-off core only
1	0	Gate-off PIXC, DMAC0 and USB
1	1	Gate-off ALL - core, PIXC, DMAC0 and USB

Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers

The handshake MDMA initial block count registers (HMDMAx_BCINIT), shown in Figure 7-21, hold the number of transfers to complete per edge of the DMARx control signal.

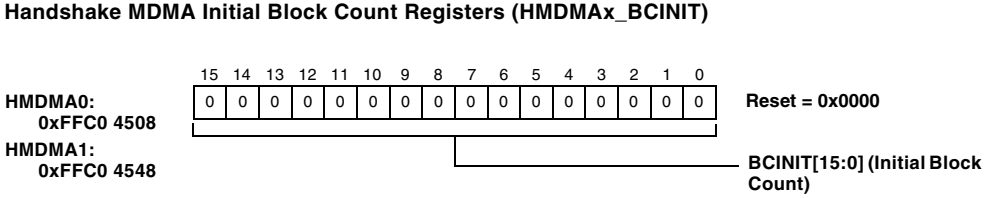


Figure 7-21. Handshake MDMA Initial Block Count Registers

Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers

The handshake MDMA current block count registers (HMDMAx_BCOUNT), shown in Figure 7-22, hold the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0x0000 = 0 transfers remaining
- 0xFFFF = 65535 transfers remaining

The BCOUNT field is loaded with BCINIT when ECOUNT is greater than 0 and BCOUNT is expired (0). Also, if the RBC bit in the HMDMAx_CONTROL register is written to a 1, BCOUNT is loaded with BCINIT. The BCOUNT field is decremented with each MDMA grant. It is cleared when HMDMA is disabled.

A block done interrupt is generated when BCOUNT decrements to 0. If the MBDI bit in the HMDMAx_CONTROL register is set, the interrupt is suppressed until ECOUNT is 0. Note if BCINIT is 0, no block done interrupt is generated, since no DMA requests were generated or grants received.

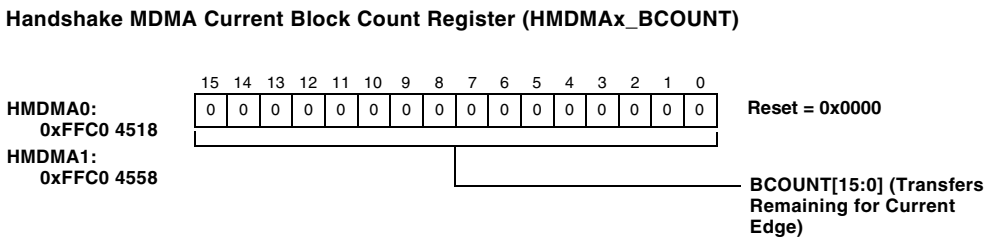


Figure 7-22. Handshake MDMA Current Block Count Registers

Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers

The handshake MDMA current edge count registers (HMDMAx_ECOUNT), shown in Figure 7-23, hold a signed number of edges remaining to be serviced. This number is in a signed, two's-complement representation. An edge is detected on the respective DMARx input. Requests occur if this count is greater than or equal to 0, and BCOUNT is greater than 0.

When the handshake mode is enabled, ECOUNT is loaded and the resulting number of requests is:

```
Number of edges + N,
```

where N is the number loaded from ECINIT. The number N is a positive or negative signed number.

Examples:

- 0x7FFF = 32767 edges remaining
- 0x0000 = 0 edges remaining
- 0x8000 = –32768: ignore the next 32768 edges

Each time that BCOUNT expires, ECOUNT is decremented and BCOUNT is reloaded from BCINIT. When a handshake request edge is detected, ECOUNT is incremented. The ECOUNT field is cleared when HMDMA is disabled.

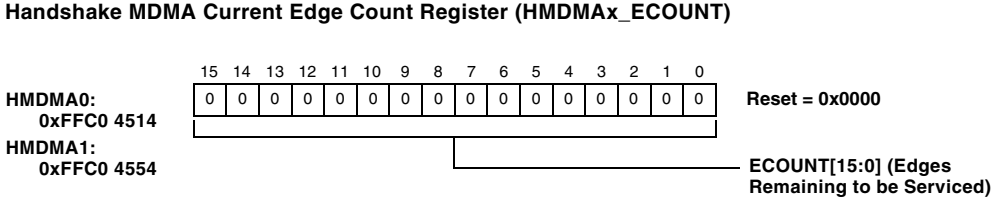


Figure 7-23. Handshake MDMA Current Edge Count Registers

Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers

The handshake MDMA initial edge count registers (HMDMAx_ECINIT), shown in Figure 7-24, hold a signed number that is loaded into current edge count (HMDMAx_ECOUNT) when the handshake DMA is enabled. This number is in a signed, two's-complement representation.

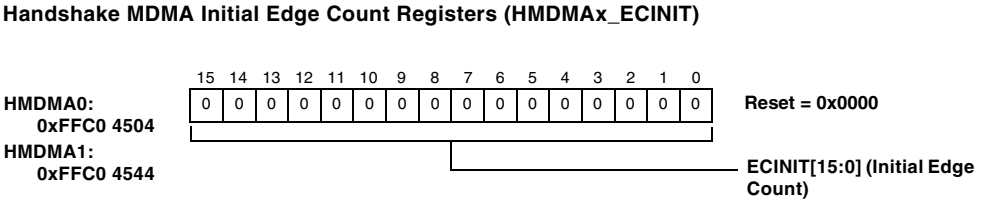


Figure 7-24. Handshake MDMA Initial Edge Count Registers

Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers

The handshake MDMA edge count urgent registers s (HMDMAx_ECURGENT), shown in Figure 7-25 and, hold the urgent threshold. If the ECOUNT field in the handshake MDMA edge count register is greater than this threshold, the MDMA request is urgent and might get higher priority.

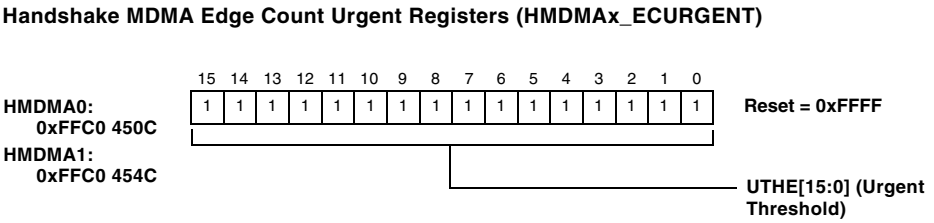


Figure 7-25. Handshake MDMA Edge Count Urgent Registers

Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers

The handshake MDMA edge count overflow interrupt registers, (HMDMAx_ECOVERFLOW), shown in Figure 7-26, hold the interrupt threshold. If the ECOUNT field in the handshake MDMA edge count register is greater than this threshold, an overflow interrupt is generated.

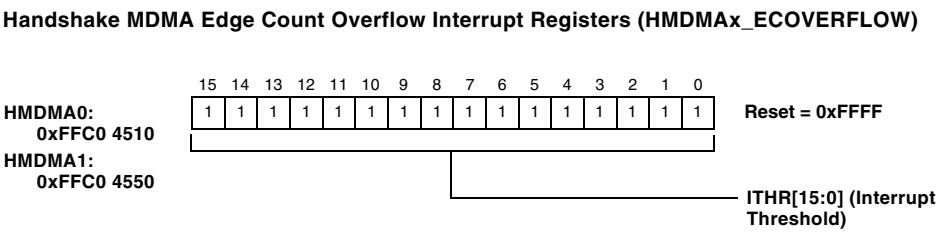


Figure 7-26. Handshake MDMA Edge Count Overflow Interrupt Registers

DMA Traffic Control Registers

The DMACx_TCPER registers and the DMACx_TCCNT registers work with other DMA registers to define traffic control.

Traffic control works within one DMA controller (DMAC0 or DMAC1), not between DMA controllers.

Table 7-23. DMA Traffic Control Registers

Register Name	Refer to	Memory-Mapped Address
DMAC0_TCPER	Listing on page 7-119	0xFFC0 0B0C
DMAC0_TCCNT	Listing on page 7-119	0xFFC0 0B10
DMAC1_TCPER	Listing on page 7-119	0xFFC0 1B0C
DMAC1_TCCNT	Listing on page 7-119	0xFFC0 1B10

This section also describes the DMAC1_PERIMUX register on page 7-121.

DMA Traffic Control Counter Period (DMACx_TCPER) Registers

The DMA traffic control counter period registers (DMACx_TCPER) are shown in Figure 7-27.

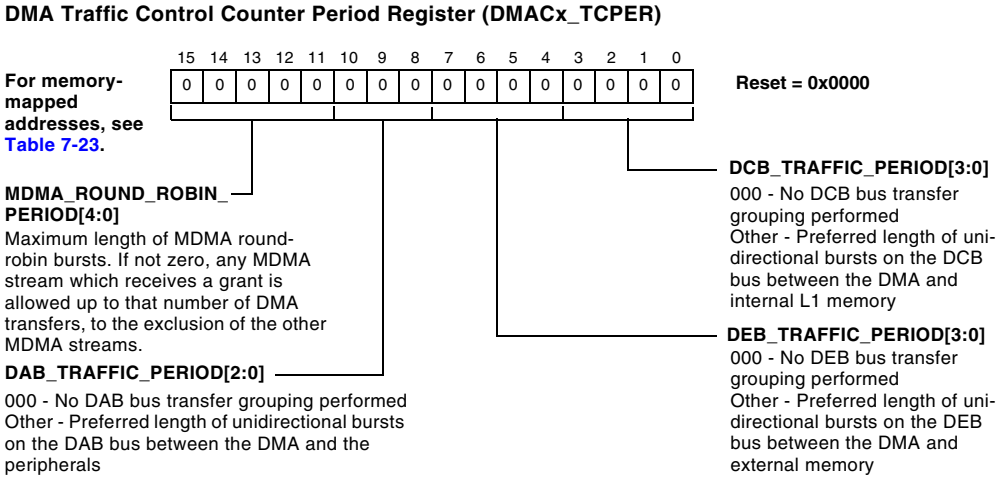


Figure 7-27. DMA Traffic Control Counter Period Registers

DMA Traffic Control Counter (DMACx_TCCNT) Registers

The DMA traffic control counter registers (DMACx_TCCNT) are shown in Figure 7-28.

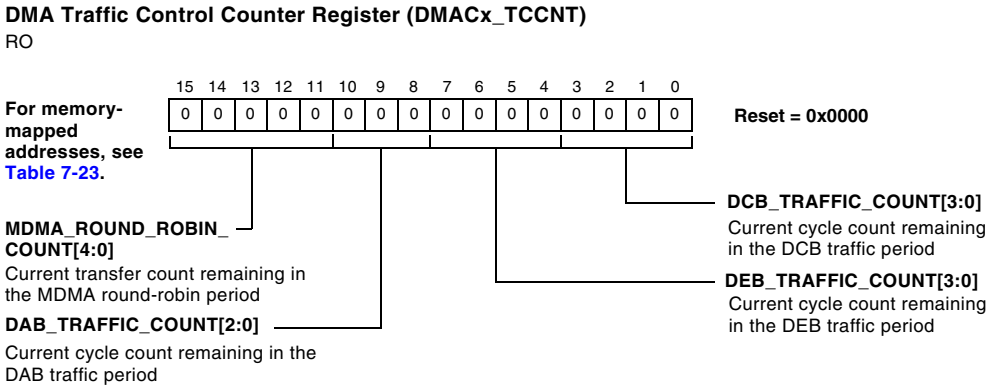


Figure 7-28. DMA Traffic Control Counter Registers

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round-robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMACx_TCPER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMACx_TCPER is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever DMACx_TCPER is written, or whenever the DEB bus changes direction or becomes idle. It then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMACx_TCPER is written, or whenever the DCB bus changes direction or becomes idle. It then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from DCB_TRAFFIC_PERIOD to begin a new burst.

DMA Controller 1 Peripheral Multiplexer (DMAC1_PERIMUX) Register

The DMAC1_PERIMUX register is shown in Figure 7-29.

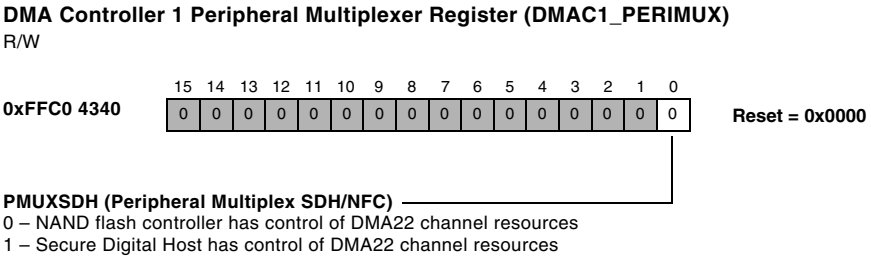


Figure 7-29. DMA Controller 1 Peripheral Multiplexer Register

The DMAC1_PERIMUX register controls the common sharing of a single DMA channel between the NAND flash controller (NFC) and the secure digital host (SDH) module. The sharing of this resource prevents the simultaneous use of the NFC and the SDH with DMA access to internal and external memory. DMAC1_PERIMUX controls the peripheral that gains access to DMA resources. DMAC1_PERIMUX is a 16-bit wide register and requires 16-bit access.

Programming Examples

The following examples illustrate memory DMA and handshaked memory DMA basics. Examples for peripheral DMAs can be found in the respective peripheral chapters.

Register-Based 2D Memory DMA

Listing 7-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel re-sorts elements of the two-dimensional data array. See Figure 7-30.

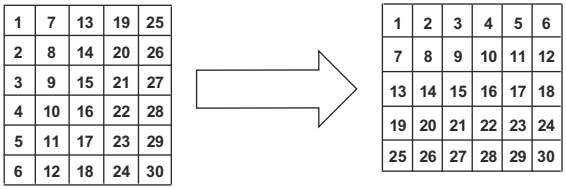


Figure 7-30. DMA Example, 2D Array

The two arrays reside in two different L1 data memory blocks. However, the arrays could reside in any internal or external memory, including L1 instruction memory, memory other than L1, and DDR SDRAM. For the case where the destination array resides in DDR SDRAM, it is a good idea to let the source channel re-sort elements and to let the destination buffer store linearly.

Listing 7-1. Register-Based 2-D Memory DMA

```
#include <defBF54x.h>
#define X 5
#define Y 6

.section L1_data_a;
.byte2 aSource[X*Y] =
    1,  7, 13, 19, 25,
    2,  8, 14, 20, 26,
    3,  9, 15, 21, 27,
    4, 10, 16, 22, 28,
    5, 11, 17, 23, 29,
    6, 12, 18, 24, 30;

.section L1_data_b;
.byte2 aDestination[X*Y];
```

```
.section L1_code;
.global _main;
_main:
    p0.l = lo(MDMA_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 7-2 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 7-2. 2-D Memory DMA Setup Example

```
memdma_setup:
    [--sp] = r7;
/* setup 1D source DMA for 16-bit transfers */
    r7.l = lo(aSource);
    r7.h = hi(aSource);
    [p0 + MDMA_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 7-3 polls the DMA status rather than using interrupts, which is the normal case in a real application.

Listing 7-3. Polling DMA Status

```
memdma_wait:
    [--sp] = r7;
memdma_wait.test:
    r7 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r7, bitpos(DMA_DONE));
    if !CC jump memdma_wait.test;
    r7 = DMA_DONE (z);
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_wait.end:
```

Initializing Descriptors in Memory

Descriptor-based DMAs expect the descriptor data to be available in memory by the time the DMA is enabled. Often, the descriptors are programmed by software at run time. Many times, however, the descriptors—or at least large portions of them—can be static and therefore initialized at boot time. How to set up descriptors in global memory depends heavily on the programming language and the toolset used. The following examples show how this is best performed in the CCES or VisualDSP++ tools' assembly language.

Listing 7-4 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 7-4. 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 CCES or VisualDSP++ tools takes advantage of C-style structures in global header files. The header file `descriptor.h` could look like Listing 7-5.

Listing 7-5. 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, 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 7-6.

Listing 7-6. 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 7-7 demonstrates a large list of descriptors that provide flow stop mode configuration. Consequently, the DMA stops by itself as soon as the work unit has finished. Software triggers the next work unit by simply writing the proper value into the DMA configuration registers. Since these values instruct the DMA controller to fetch descriptors in large list mode, after being started, the DMA immediately fetches the descriptor and then overwrites the configuration value again with the new settings.

Note the requirement that source and destination channels stop after the same number of transfers. In between stops the two channels can have completely individual structure.

Listing 7-7. Software-Triggered Descriptor Fetch

```
#define N 4
.import "descriptor.h";
.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:
```

Handshake Memory DMA Example

The functional block for the handshake MDMA operation can be seen completely separately from the MDMA channels themselves. Therefore the following HMDMA setup routine can be combined with any of the MDMA examples discussed above. Be sure that the HMDMA module is enabled before the MDMA channels.

Listing 7-8 enables the HMDMA1 block which is controlled by the DMAR1 pin and is associated with the MDMA1 channel pair.

Listing 7-8. HMDMA1 Block Enable

```
/* optionally, enable all four bank select strobes */
    p1.l = lo(EBIU_AMGCTL);
    p1.h = hi(EBIU_AMGCTL);
    r0.l = 0x0009;
    w[p1] = r0;

/* function enable for DMAR1 */
    p1.l = lo(PORTH_FER);
    r0.l = PH6;
    w[p1] = r0;
    p1.l = lo(PORTH_MUX);
    r0.l = lo(MUX6_1);
    r0.h = hi(MUX6_1);
    [p1] = r0;

/* every single transfer requires one DMAR1 event */
    p1.l = lo(HMDMA1_BCINIT);
    r0.l = 1;
    w[p1] = r0;
```

```
/* start with balanced request counter */
    p1.l = lo(HMDMA1_ECINIT);
    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 intent is to copy from internal memory to external devices, the above setup is appropriate. It controls the Memory DMA's destination channel. If the intent is to read data from external memory, set the SND bit in the HMDMAx_CONTROL register to control the source channel instead.

8 HOST DMA PORT

This chapter describes the Host DMA port (HOSTDP) and includes the following sections:

- "Overview" on page 8-1
- "Interface Overview" on page 8-3
- "Description of Operation" on page 8-3
- "Programming Model" on page 8-21
- "Host DMA Port Registers" on page 8-24
- "Programming Examples" on page 8-30

Overview

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

Pay particular attention to nomenclature for the Host DMA port (HOSTDP). All register and pin names have a `HOST_` prefix. The HOSTDP is a peripheral on the ADSP-BF54x processor processor, which is referred to as the slave processor or Blackfin slave. The host processor is also referred to as the host, master, external host, or external master.

When using one of the HOSTDP boot modes, the boot kernel does not disable the HOSTDP module or the associated DMA channels when the boot completes.

The HOSTDP is enabled through the peripheral access bus (PAB) interface. Once enabled, the DMA is controlled by an external host. The external host can then program the DMA to send/receive data to any valid internal or external memory location.

Features

The HOSTDP controller includes the following features:

- External master to configure DMA READ/WRITE data transfers and read port status
- Flexible asynchronous memory protocol for external interface
- 8/16-bit external data interface to host device
- Half-duplex operation
- Little/big endian data transfer
- Internal FIFO which holds sixteen 32-bit words
- Acknowledge mode allows flow control on host transactions
- Interrupt mode guarantees a burst of FIFO depth host transactions
- Ability to enable and disable data reads/writes
- DMA bandwidth control

Interface Overview

Table 8-1 defines the pins for the HOSTDP interface. The interface uses a flexible asynchronous memory interface, which can be gluelessly connected to a variety of host processors.

Table 8-1. HOSTDP External Pins

Pin	Description
Port D - HOST_DATA <15:0>	16-bit data port
PG5- $\overline{\text{HOST\_CE}}$	Chip enable for the HOSTDP
PG7 - $\overline{\text{HOST\_WR}}$	Write strobe
PG6- $\overline{\text{HOST\_RD}}$	Read strobe
PH3 - HOST_ADDR	Address pin 0: data port access 1: configuration port access
PH4 - HOST_ACK (HRDY/FRDY)	Flow control pin: HRDY-acknowledge mode and FRDY- interrupt mode

Due to the Blackfin processor's use of multiplexed pins, utilizing the Host DMA port can preclude the use of other peripherals. EPP2 is unavailable when using the HOSTDP, and EPP1 can be used in 8-bit mode if the HOSTDP is also in 8-bit mode. Refer to "General-Purpose Ports" on page 9-1 for a complete description of the pin multiplexing scheme.

Description of Operation

The following sections describe the operation of the HOSTDP interface.

Architecture

The HOSTDP block diagram, shown in Figure 8-1, illustrates the overall architecture of the HOSTDP.

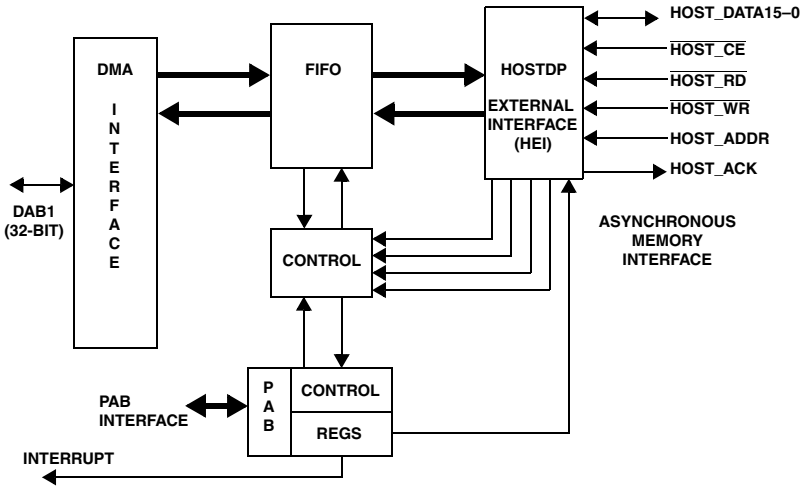


Figure 8-1. HOSTDP Block Diagram

The HOSTDP is enabled/disabled through PAB writes to the HOST_CONTROL register. Once enabled, the HOSTDP interfaces to the external world using asynchronous memory protocol and handshakes with the DMA controller internally using the DMA access bus (DAB). The HOSTDP allows the external host to program the DMA to transfer data in either direction. The HOSTDP can be broken into five functional blocks, identified as follows:

- **Host External Interface (HEI)** This block interfaces to the external host and based on inputs from the host device initiates data or control message transfers.

- **PAB Interface** The HOSTDP is programmed/queried for status by reads or writes to appropriate registers in this block through the PAB.
- **FIFO** A dual-port FIFO is used for data transfers and can store up to sixteen 32-bit words.
- **Control** The control block handles the HOSTDP's different states as well as the handshakes between the external host device and DMA interfaces.
- **DMA Interface** This block is connected to the DAB and interacts with the DMA to transfer control messages and data between DMA and external host device.

Functional Description

The following sections describe the functional operation of the Host DMA port (HOSTDP).

HOSTDP Configuration

Before any data transfer can occur, the DMA engine must be configured by the host processor. Because the host is unaware of the internal state of the Host DMA port peripheral and its associated DMA activity, the host processor is required to check the `ALLOW_CNFG` bit in `HOST_STATUS` register before attempting configuration writes. Additionally, this status read sets some internal states inside the Host DMA port. Configuration requires seven 16-bit words to be written in the following order to the configuration port before host read data or host write data operations can occur:

- `HOST_CONFIG`
- `START_ADDR.L`
- `START_ADDR.H`

- XCOUNT
- XMODIFY
- YCOUNT
- YMODIFY

The only word different from the standard DMA described in Chapter 7, "Direct Memory Access" is the HOST_CONFIG word. Each bit is described there. Refer to Figure 8-2 for a description.

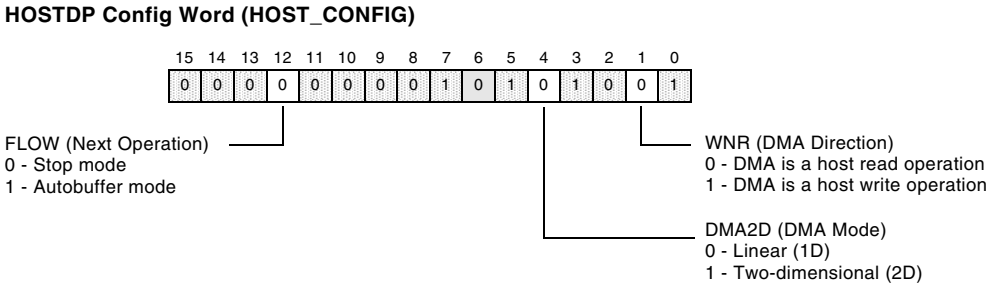


Figure 8-2. HOSTDP Configuration Word

Additional information for the HOST_CONFIG bits includes the following:

- **Host DMA Direction** (WNR)
  If this bit is written high, DMA writes to memory (host write). If this bit is written low, DMA reads from memory (host read).
- **Host DMA Mode** (DMA2D)
  If written low, it is linear one dimensional (1D) DMA. If written high, it is two-dimensional mode (2D).
- **FLOW** (FLOW)
  When this bit is cleared, the DMA runs in STOP mode. When this bit is set, the DMA runs in AUTOBUFFER mode.

For information on how these words are used to configure the DMA, refer to "Direct Memory Access" on page 7-1.

Before accessing the data port, the host processor must write all seven descriptor words. The HOSTDP module does not forward descriptors to the DMA channel until it has received all seven words. Similarly, the host processor is not permitted to provide new descriptor data before all data words of the former work unit are transferred. However, the host can truncate an initiated transfer using the DMA_FINISH control command. As always, ALLOW_CNFG in the HOST_STATUS register must be polled before writing a new configuration to the Host DMA port. See "Control Commands Between the External Host and HOSTDP" on page 8-20 for additional information.

Additional latency is incurred when a host read data operation follows a host write data operation. Even though the configuration for the host read is complete, the DMA engine must first empty the FIFO for the host write operation and then change directions and start filling the FIFO for the host read data operation.

HOSTDP Transactions

The HOSTDP is enabled by writing to the HOSTDP_EN bit of the HOST_CONTROL register. In order to disable the HOSTDP, the HOSTDP_RST bit must be asserted before clearing HOSTDP_EN. There are four types of host port transactions. Each type of access is controlled by the HOST_ADDR and whether the $\overline{\text{HOST\_RD}}$ or $\overline{\text{HOST\_WR}}$ signal is asserted.

When chip enable ($\overline{\text{HOST\_CE}}$) is inactive, the HOSTDP stays idle. Modes listed in Table 8-2 are only possible when $\overline{\text{HOST\_CE}}$ is active.

Table 8-2. Types of Host Port Transactions

Address	HOST_RD	HOST_WR	HOST_CE	Function
0x0	0	1	0	Host read data operation
0x0	1	0	0	Host write data operation
0x1	1	0	0	Host write configuration or control command
0x1	0	1	0	Host read HOST_STATUS register

Host Read Status

The Host DMA port is robust against on-the-fly changes of data direction. However, in acknowledge mode, it is encouraged not to initiate a new work unit with different data direction before the FIFOEMPTY bit in the HOST_STATUS register is cleared. This is to avoid excessive wait states inserted by HRDY.

The external host can read the HOST_STATUS register at any time. By performing this operation, the external host can query the status of the HOSTDP. Note that in 8-bit configurations, the host can only read the lower byte of the HOST_STATUS register. HOST_STATUS can also be read through a PAB access. When accessed through the PAB, all 16 bits of HOST_STATUS are always read. The contents of HOST_STATUS are detailed in "Host DMA Port Registers" on page 8-24.

Host Read Data and Host Write Data Operations

After the HOSTDP is configured and enabled by way of PAB accesses and the DMA channel is configured through host write configuration accesses, data can be transferred.

All DMAs between the HOSTDP FIFO and memory are 32-bit transactions. This is important for setting XMODIFY and YMODIFY. The amount of data moved between the host processor and the HOSTDP must be a multiple of the FIFO depth (sixteen 32-bit

words). The user is required to set the XCOUNT/YCOUNT values and to also ensure that the correct number of host data reads or host data writes are performed.

A host write data operation is used to transfer data from the host to the slave processor. The host performs write transactions and the HOSTDP writes the data from these transactions into its FIFO. The DMA engine concurrently moves data from the HOSTDP's FIFO to the location in memory specified by the DMA configuration words.

A host read data operation is used to transfer data from the slave processor to the host. The DMA engine moves data from the specified location in the Blackfin processor slave's memory into the HOSTDP's FIFO. The host performs read accesses to read data out of this FIFO.

In the case of host writes, the host processor must "pad" the end of the transfer with dummy data to ensure this (for example, if the host wants 31 words it must send an extra dummy word to equal 32). In the case of host reads, dummy reads must be performed at the end and the host can then throw away the results. This is true in both interrupt mode and acknowledge mode.

Since all DMAs from the HOSTDP are 32-bits, data is packed into 32-bit words in the HOSTDP FIFO on host data write operations. Data (32-bit) in the FIFO is unpacked into 8-bit or 16-bit words (depending on the HOSTDP_DATA_SIZE setting in HOST_CONTROL) for transmission during host data read operations. Because all DMAs are 32-bits and the data bus is either 8-bits or 16-bits, the total of XCOUNT * YCOUNT is 1/4 (8-bit) or 1/2 (16-bit) the number of data reads or writes the host processor performs.

HOSTDP Modes of Operation

There are two modes of flow control in the HOSTDP: acknowledge mode and interrupt mode. These two modes provide flow control between the host and the slave processor by way of a single hardware signal pin. This signal has different names depending upon the mode of operation. The flow control mode is configured by the slave processor when enabling the HOSTDP (see HOST_CONTROL register).

In acknowledge mode, the signal is called HRDY and is used to add wait states to a host transaction when the HOSTDP is not ready to transfer data. The HRDY signal is level-sensitive.

In interrupt mode, the signal is called FRDY and is used as an edge-triggered signal. This signal is connected to the host as an interrupt input. A falling edge on it signals the host that the HOSTDP is ready for a guaranteed FIFO depth number of back-to-back transactions. For host write operations, this occurs when the FIFO is empty. For host read operations, this occurs when the FIFO is full.

Acknowledge Mode

For host data write operations, HRDY negates when the FIFO is full, thereby inserting wait states. As soon as the DMA engine moves data out of the FIFO, HRDY asserts, indicating to the host the host data write operation is complete.

For host data read operations, HRDY will negate when the FIFO is empty, thereby inserting wait states. As soon as the DMA engine moves data into the FIFO, HRDY asserts indicating to the host the host data read operation is complete.

The HRDY signal must be pulled high by an external pull-up resistor by default at power-up/reset and when the HOSTDP is not enabled. HRDY is only driven when $\overline{\text{HOST\_CE}}$ is asserted low.

When the host is performing a host write configuration or HOST_STATUS reads, HRDY always remains asserted and no wait states are added.

Acknowledge Mode Timing Diagrams

This section gives further details on the HOSTDP timings for acknowledge mode. The host processor must follow these rules on every bus cycle independent of the nature of the access and the status of slave processor.

(It is assumed that the Blackfin slave processor has booted and the HOSTDP is functional.)

As discussed in the following section, HRDY has an external pull-up register:

1. If $\overline{\text{HOST\_CE}}$ is high, HRDY is three-stated (not driven).
2. HRDY is driven by the slave processor only when $\overline{\text{HOST\_CE}}$ is asserted low by the external host device.
3. If $\overline{\text{HOST\_CE}}$ as well as either $\overline{\text{HOST\_RD}}$ or $\overline{\text{HOST\_WR}}$ are asserted and HOST_ADDR is high (configuration port access or status read), HRDY remains driven high (READY).
4. If $\overline{\text{HOST\_CE}}$ as well as either $\overline{\text{HOST\_RD}}$ or $\overline{\text{HOST\_WR}}$ are asserted, and HOST_ADDR is low (data port access), one of two things happen:
    a. If $\overline{\text{HOST\_RD}}$ is asserted and the desired FIFO data can be transferred on the data bus pins within time T, HRDY remains driven high. If $\overline{\text{HOST\_WR}}$ is asserted and if the data can be stored in the FIFO within time T, HRDY remains high.
    b. If the desired FIFO data cannot be transferred on the data bus pins or stored in the FIFO within time T, HRDY is driven low quickly. At some later time after the FIFO data can be transferred, HRDY is driven high.

The two timing diagrams, shown in Figure 8-3 and Figure 8-4, are necessary to understand the function of HRDY.

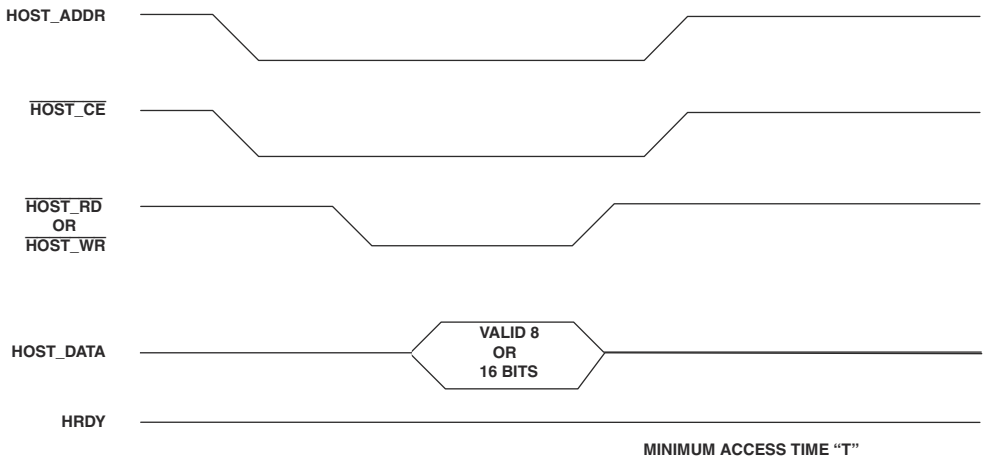


Figure 8-3. No Delay in Host Bus Cycle

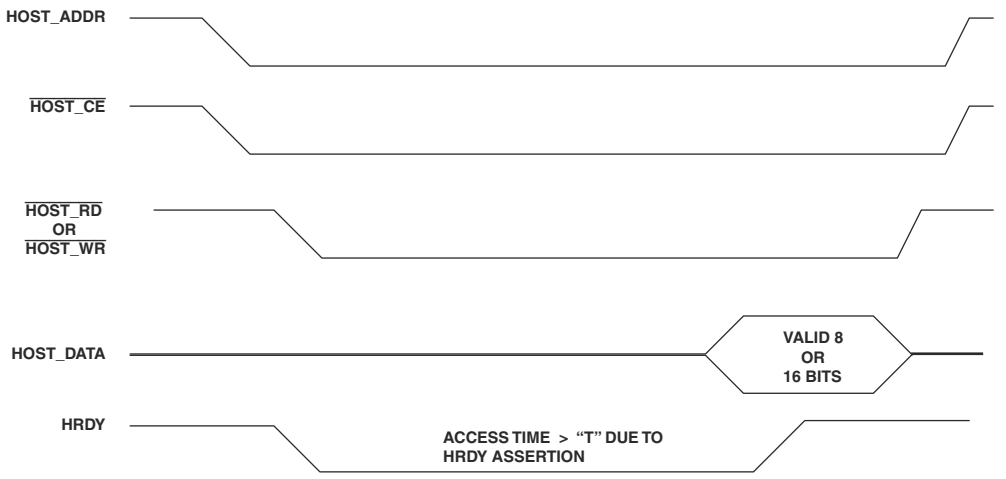


Figure 8-4. Delay in Host Bus Cycle Caused by HRDY

Host Bus Timeout

In acknowledge mode, an optional host bus timeout feature is implemented as a mechanism to alert the host when a programmed period of time has expired during a host read/write data transaction and the HOSTDP is still unable to complete the transaction with HRDY assertion. This condition can occur when the internal shared DMA bus has a lot of traffic from other peripherals on it. (This situation should never occur in a working system, but could occur if a mistake was made in software. An example is mistakenly disabling the DMA channel while the HOSTDP is attempting to transfer data.) An internal timer is started when $\overline{\text{HOST\_CE}}$ and either $\overline{\text{HOST\_RD}}$ or $\overline{\text{HOST\_WR}}$ are asserted. The timer is reset whenever HRDY is asserted.

The feature can be enabled by the BT_EN bit in the HOST_CONTROL register. When enabled, the HOSTDP generates an interrupt when a pre programmed timeout value set in the HOST_TIMEOUT register expires. In a typical application, the interrupt service routine toggles a GPIO pin which is connected to the host processor to alert it of this condition. Additionally, the interrupt service routine can perform writes to the HOST_CONTROL register to perform the following:

- Stop the DMA channel by clearing the DMAEN bit in the DMAx_CONFIG register
- Write the HRDY_OVR bit in the HOST_CONTROL register to assert the HRDY pin to allow the host bus cycles to continue while the host is being signaled of this condition by way of a GPIO pin
- Disable the HOSTDP by clearing the HOSTDP_EN bit in the HOST_CONTROL register

The actual timeout value can be programmed in the HOSTDP_TOUT register.

Because it is important for the host to be aware that a timeout condition occurred, it is required that the host processor read the HOST_STATUS register and check the HOSTDP_TOUT bit. The ADSP-BF54x processor slave

processor reads the actual bit, allowing it to take the timeout interrupt, and write-one-to-clear the HOSTDP_TOUT bit. The host processor reads a special shadow version of this bit which remains set until the host has read it or a hard reset occurs.

Interrupt Mode

The FRDY signal acts as an edge-sensitive (high-to-low transition) signal to provide an interrupt to the external host to indicate when data transfer can proceed. The interrupt provided by the slave processor to the external host device by way of the FRDY signal is used to indicate the status of the Host DMA port's FIFO. Host data read and host data write accesses are described next. The host device always masters the transactions and the Blackfin processor is always a DMA slave device.

In interrupt mode, the FRDY signal always is driven by the slave processor and does not require an external pull-up resistor.

For host write operations, the FRDY signal transitions from high to low whenever the FIFO is empty, causing an interrupt to the host to tell it to write to HOSTDP. The host can then perform a buffer depth number of write cycles to fill the FIFO. During these writes, the FRDY signal transitions high again, but this is ignored by the host. After the FIFO's contents have been moved to memory by the DMA engine, the FIFO becomes empty. At this time, FRDY will once again transition from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

For host read operations, the FRDY signal transitions from high to low whenever the FIFO is full, causing an interrupt to the host to tell it to read from the HOSTDP. The host can then perform a buffer depth number of read cycles to empty the FIFO. During these reads, the FRDY signal transitions high again, but this is ignored by the host. The DMA engine fills the FIFO from data in memory. Once the FIFO becomes full again, the FRDY

signal once again transitions from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

In interrupt mode, the FRDY signal always reflects the status of the FIFO. For host configuration writes or host reads of HOST_STATUS, accesses always meets the minimum cycle time T and the FRDY signal is not used for flow control of these accesses.

Figure 8-5 shows the timing of the interrupt mode transactions. The total number of words in the transfer are divided into blocks that contain a FIFO depth's number of words. These blocks are transferred whenever a high-to-low transition occurs on the FRDY signal.

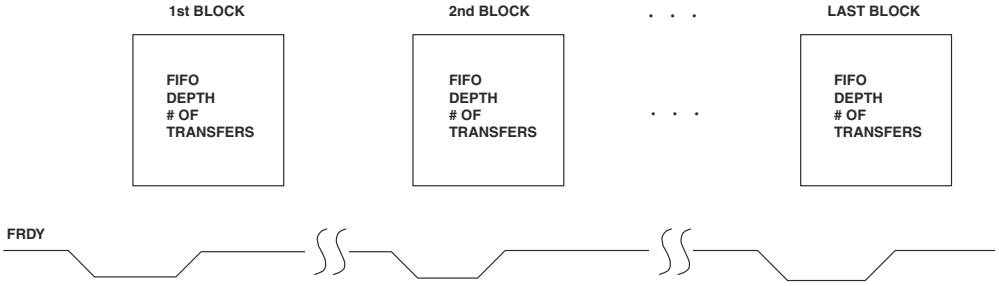


Figure 8-5. Interrupt Mode Bus Cycles

DMA STOP Mode and AUTOBUFFER Mode

The FLOW bit in HOST_CONFIG word controls whether the DMA channel runs in stop mode or autobuffer mode.

In stop mode, the DMA performs a block transfer once as programmed by the HOST_CONFIG, XCOUNT/YCOUNT, XMODIFY/YMODIFY, START_ADDR.L/H registers. To perform another block transfer requires the host to reconfigure these parameters. For stop mode, the interrupt service routine is required

to set the DMA_CMPLT bit in the HOST_STATUS register. This prepares the HOSTDP for the next transfer. The host is not required to poll the DMA_CMPLT bit before starting a new work unit.

In autobuffer mode, the DMA performs continuous block transfers based on the parameters programmed by the HOST_CONFIG, XCOUNT/YCOUNT, XMODIFY/YMODIFY, START_ADDR.L/H registers. Once the number of words specified by XCOUNT/YCOUNT are transferred, the DMA engine sets its address pointer back to START_ADDR.L/H and performs another block transfer. For autobuffer mode, the interrupt service routine should only set the DMA_CMPLT bit in the HOST_STATUS register when it wishes to complete the transfers. After this bit is set, the HOSTDP block expects to be reprogrammed with a new set of DMA register values.

Bus Widths and Endian Order

The HOSTDP can be programmed to be 16-bits wide or 8-bits wide. Additionally, the byte order can be programmed as little endian or big endian. All ensuing data and configuration transactions with the host occur in the programmed endianess setting.

For 16-bit transfers, shown in Figure 8-6, the upper and lower bytes are based on the big/little endian setting. When set to little endian, the order of the bytes on the HOST_DATA[15:0] bus is unchanged. For big endian, the upper and lower bytes of HOST_DATA[15:0] are swapped before being stored internally.

For 8-bit transfers, the order in which the bytes are sent are based on the bit/little endian setting as shown in Figure 8-7. Consider a 16-bit word stored in internal memory:

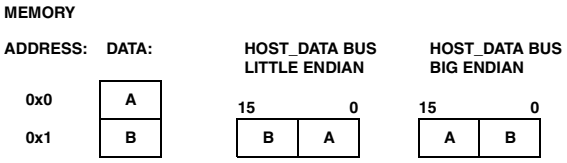


Figure 8-6. 16-Bit Transfer Byte Order

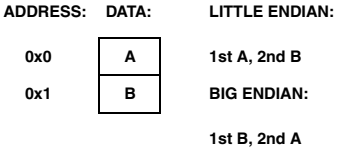


Figure 8-7. 8-Bit Transfer Byte Order

Access Control

Configurations only occur when they are allowed. The ALLOW_CNFG bit does not go low after configuration words are written if the access type is disallowed. In the case of a disallowed configuration, the configuration words are not driven on the DAB bus, and DMA controller does not get programmed. There is no NACK provided to the host in the event of a disallowed configuration.

By default, the HOSTDP module prohibits the external host from performing host data read and writes. Blackfin software is required to enable host reads or writes. Host data reads and writes are enabled or disabled separately by the EHR and EHW bits in the HOST_CONTROL register. Once enabled, the host can perform read or write transactions. Writes to the configuration port, control commands and status reads are permitted regardless of the EHR and EHW settings. It is very important that the EHR/EHW bits are written once before ever allowing configuration from the Host and then not changed later.

For more information, see the memory configuration discussion in "Security" on page 16-1.

In acknowledge mode, if the transactions are disabled, host writes are still allowed on the bus, but the actual write data is ignored. Similarly, host reads still occur on the bus, but the data returned is indeterminate.

In interrupt mode, transitions on FRDY never occur.

The host cannot interrogate the HOSTDP to see whether only read or write access is granted. Therefore, keep the EHR and EHW settings global without altering them.

Improving HOSTDP DMA Bus Bandwidth

Since the HOSTDP can be configured as a 16-bit wide parallel interface, data can move into and out of the peripheral quickly as compared to other serial peripherals on the chip. A FIFO is used to buffer this data and internal DMA bus requests are made judiciously to minimize the amount of DMA bandwidth that is used on the DMA bus. DAB bus arbitration overhead and direction change penalties are minimized. This is the default behavior (BDR=1 in HOST_CONTROL) and the Host DMA port generally follows this behavior, shown in Table 8-3, for receive (host write) operations:

Table 8-3. Host Write Operations

32-Bit Words in FIFO	DMA Request Freq (SCLK cycles)	Bursts per DMA Request
1 – 4	24	Up to 4
5 – 8	16	4
9 – 12	8	4
>12	2	0

For example, if there are ten words written into the FIFO by the host processor, on the eighth SCLK cycle, DMA is requested. Once the DAB approves the request, it transfers four words. Assuming the host processor does not write any new words to the FIFO, the HOSTDP again requests DMA 16 cycles later and another four words are transferred. Twenty-four SCLK cycles later, the remaining two words are transferred. Note that words stored in the FIFO are 32 bits.

For transmit (host read) operation, the values look similar. Refer to Table 8-4.

Table 8-4. Host Read Operation

32-bit Words in FIFO	DMA Request Freq (SCLK cycles)	Bursts per DMA Request
0 – 4	2	0
5 – 8	8	4
9 – 12	16	4
>12	24	Up to 4

This default behavior can be overridden by clearing the burst DMA requests (BDR) bit in the HOST_CONTROL register. This allows the HOSTDP to perform internal DMA bus requests whenever there is a single word of data in the FIFO for host writes and at least one empty slot for host reads. In this case, DMA bus requests are made more often. This allows higher throughput through the HOSTDP at the expense of the other peripherals on the chip.

Control Commands Between the External Host and HOSTDP

Control commands can be sent from the host to the HOSTDP by writing to the configuration port with bits 3 and 2 of the data high. When the Host DMA port is waiting for configuration, a control command cannot be sent because it will be misinterpreted as a configuration write. After configuration is finished, control commands can be issued at any time. If the host is unsure of whether configuration is pending, it needs to read the HOST_STATUS register to check.

The commands that are supported are shown in Table 8-5.

Table 8-5. Control Commands

HOST_DATA[7:0]	Command
8'b000111xx	Host IRQ
8'b001011xx	DMA finish
8'b001111xx to 8'b111111xx	Ignored

The host IRQ command provides a mechanism for the host to interrupt the HOSTDP. When the host writes a host IRQ command to the configuration port, the HIRQ bit in the HOST_STATUS register is set and a HOSTDP status interrupt is signaled.

The handshake bit (HSHK) in HOST_STATUS can be set or cleared anytime by the slave processor. This bit can be used as a flag which the host can read. In an application, the host might interrupt with the host IRQ command requesting information. The interrupt service routine could then set or clear the HSHK bit. The host could then read the status register and test for the value of the HSHK bit.

The DMA finish command performs all the same functions as the HOSTDP reset (HOSTDP_RST) bit in HOST_CONTROL, except modifying the HOST_STATUS register contents. In addition, it stops any DMA activity.

The DMA FINISH command may not complete right away, instead it completes only after the DAB state machine has moved to a particular idle state.

There are additional restrictions on when a DMA Finish command may be sent by the host processor. For more information see "DMA Control Commands" on page 7-39.

When the HOSTDP module receives a `FINISH` command from the host during a write operation, the DMA channel's FIFO is still drained gracefully and requests a DMA completion interrupt. However, the HOSTDP's FIFO is flushed immediately. To avoid loss of data, the host may want to wait until the `FIFOEMPTY` bit in `HOST_STATUS` is asserted before issuing the finish command.

Programming Model

The following sections describe the programming model for the Host DMA port.

ADSP-BF54x processor Slave

Figure 8-8 shows how to enable the Host DMA port. It shows how to properly set up interrupt service routines for both host read and write which clear the interrupts and prepare the HOSTDP for to be configured by the host again.

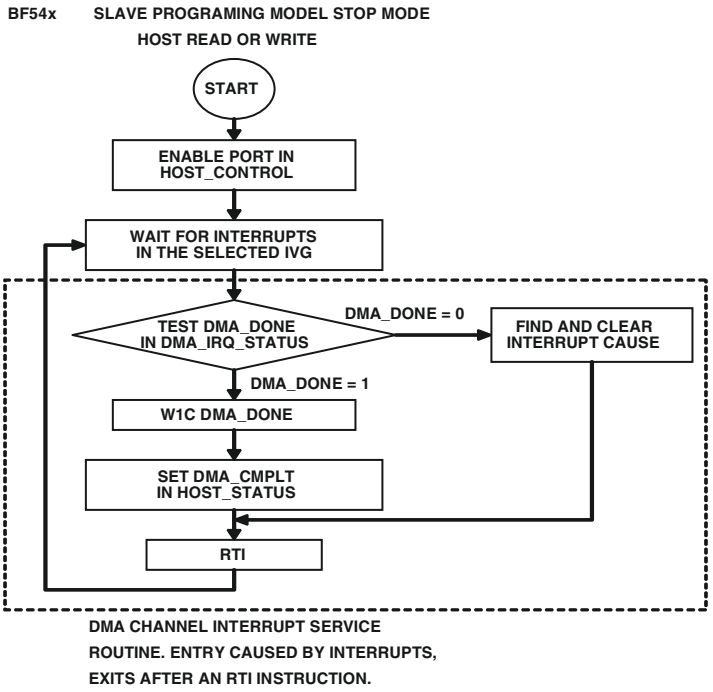


Figure 8-8. Enable the Host DMA Port

Host Processor

Figure 8-9 and Figure 8-10 demonstrate how to program a host processor to send a configuration to the ADSP-BF54x processor slave. They also show when to send or receive data in both acknowledge and interrupt modes.

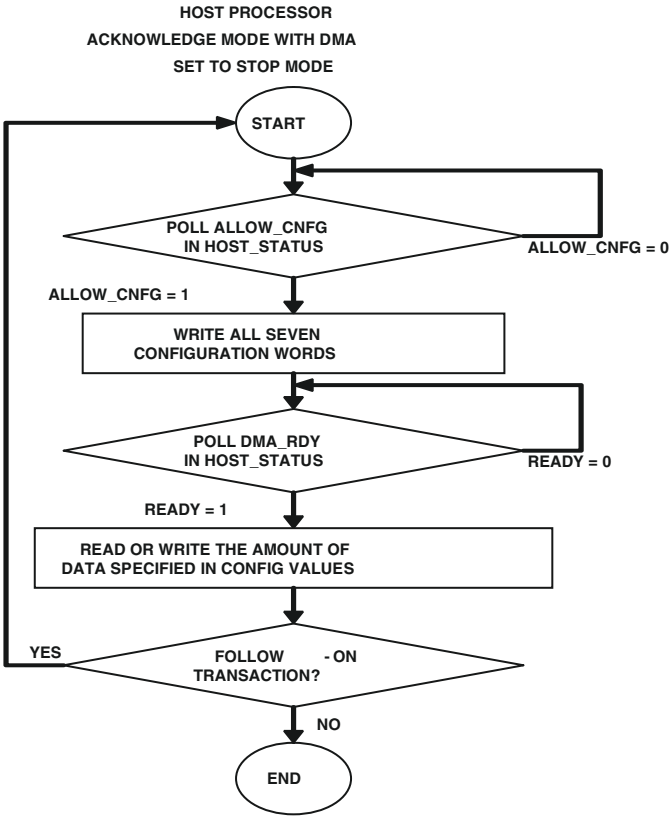


Figure 8-9. Program Host Processor, Part 1

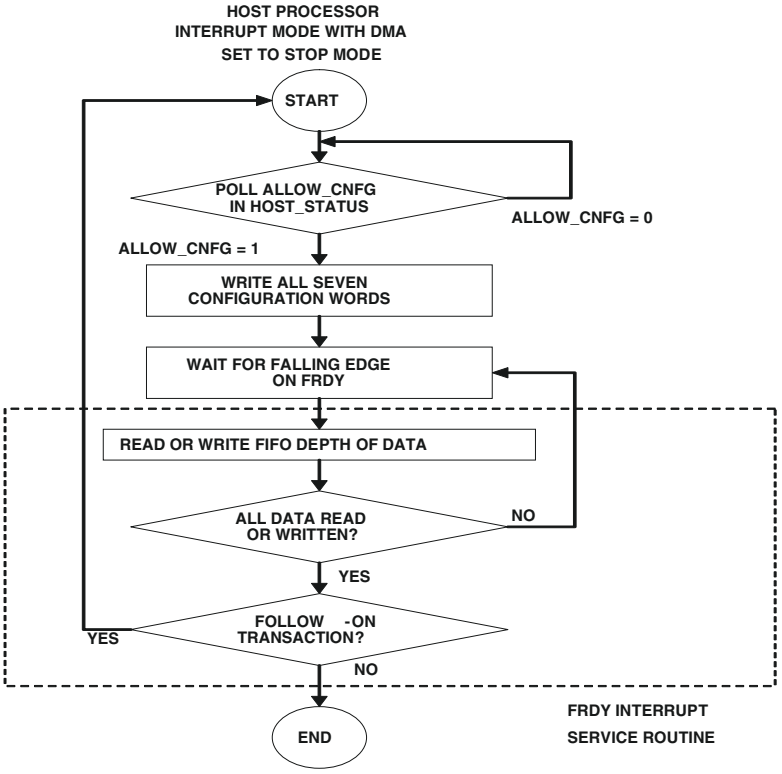


Figure 8-10. Program Host Processor, Part 2

Host DMA Port Registers

Descriptions and bit diagrams for each of the MMRs discussed in this chapter are provided in the following sections:

"Host DMA Port Control (HOST_CONTROL) Register" on page 8-25
"Host DMA Port Status (HOST_STATUS) Register" on page 8-27
"HOSTDP Timeout (HOST_TIMEOUT) Register" on page 8-29

Host DMA Port Control (HOST_CONTROL) Register

The HOSTDP control register (HOST_CONTROL), shown in Figure 8-11, is used to enable the HOSTDP module as well as to establish transfer modes of operation.

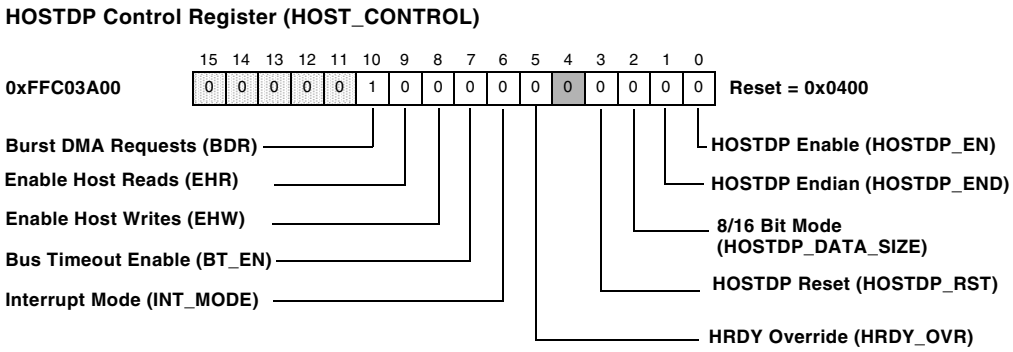


Figure 8-11. HOSTDP Control Register

Additional information for the HOST_CONTROL register bits include:

- **HOSTDP Enable** (HOSTDP_EN)
  This bit enables the HOSTDP interface. This bit controls the muxing of the shared HOSTDP and PPI pins. Before disabling the HOSTDP, always reset it first.
- **Little/Big Endian** (HOSTDP_END)
  When set, this bit swaps the lower and upper byte of data when reading or writing to HOSTDP FIFO. A value of 0 represents little endian and a value of 1 represents big endian.
- **8/16-Bit Host Data Transfer** (HOSTDP_DATA_SIZE)
  This bit sets the HOSTDP external data transfer width. This bit, along with HOSTDP_EN, controls the muxing of the HOSTDP data pins and the EPPI pins. A value of 0 is 8-bit data and a value of 1 is 16-bit

- **HOSTDP Reset** (HOSTDP_RST)
  This is a soft reset which does not affect the contents of HOST_CONTROL. Programming this bit causes the FIFO to flush, turns off the DMA channel, and returns the HOSTDP to a state where it is waiting for configuration. It also causes HOST_STATUS to clear to the same value as a hard reset with the exception of the BTE bit, which is always the same as BT_EN in HOST_CTL. Host DMA port reset will not complete right away, instead it completes only after the DAB state machine has moved to a particular idle state. This bit is always read as a binary 0.

- **Host Ready Override** (HRDY_OVR)
  Setting this bit high forces HRDY high. If HRDY_OVR bit is written high, HRDY is driven high for all remaining FIFO transfers. Also, the ALLOW_CNFG bit is driven low to prevent accidental configurations.

- **Interrupt Mode** (INT_MODE)
  This bit, when set, is used to select interrupt mode. When cleared, it selects acknowledge Mode. A value of 0 selects acknowledge mode and a value of 1 selects interrupt mode.

- **Bus Timeout Enable** (BT_EN)
  This bit, when set, enables HOSTDP's interrupt to occur when a current host transaction has not finished before a programmed timeout value occurs.

- **Enable HOSTDP Write** (EHW)
  This bit, when set, enables HOSTDP's writes to occur. If disabled, host writes appear to occur on the pins, but the actual write data is ignored.

- **Enable HOSTDP Read** (EHR)
  This bit, when set, enables HOSTDP's reads to occur. If disabled, host reads return zero data.

- **Burst DMA Requests** (BDR)
  When set, as by default, the HOSTDP's module groups multiple data words and requests DMA bursts to the DAB bus. When cleared, every individual data word requests its separate DMA transfer.

Host DMA Port Status (HOST_STATUS) Register

The HOSTDP status register (HOST_STATUS), shown in Figure 8-12, holds the key status information of the HOSTDP. Bits in this register are read by the external host to query status of transaction. This register can also be read and written through PAB. Note the differences in how to write and clear bits as well as the many bits which are read-only.

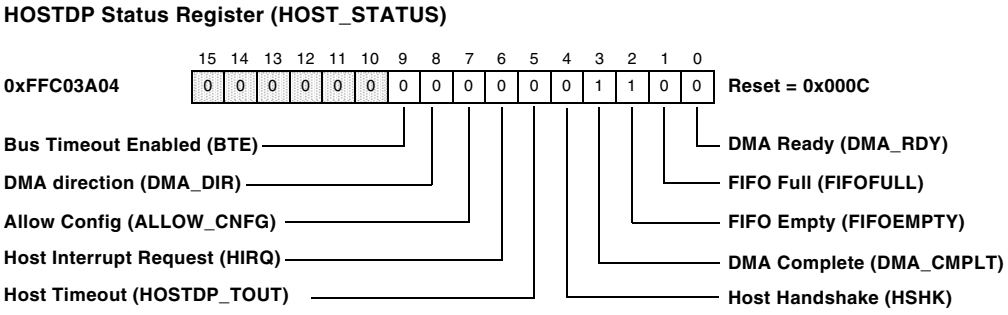


Figure 8-12. HOSTDP Status Register

Additional information for the HOST_STATUS register bits include:

- **DMA Ready** (DMA_RDY) - read-only
  This bit is set one cycle after the last control word (YMODIFY) is written to the DMA. The bit is cleared when the COMPLETE bit is set by software.
- **FIFO Full** (FIFOFULL) - read-only
  This bit is set when the HOSTDP FIFO is full.
- **FIFO Empty** (FIFOEMPTY) - read-only
  This bit is set when the HOSTDP FIFO is empty.
- **DMA Complete** (DMA_CMPLT) - write-1-to-set
  This bit must be set by software in the interrupt service routine called when the DMA operation is completed. This bit is cleared after the last control word (YMODIFY) is written to the DMA controller.
- **HOSTDP Handshake** (HSHK) - read/write
  This bit is set and cleared by software and functions as a general-purpose handshake bit. Often it is used to indicate an error to the host device. This bit does not control HOSTDP hardware and is cleared by the HOSTDP_RST bit.
- **HOSTDP Timeout** (HOSTDP_TOUT) - write-1-to-clear
  This bit is set when the HOSTDP time-out occurs. When set, it requests a HOSTDP status interrupt. The interrupt service routine (ISR) must write this bit to one to clear it.
- **HOSTDP Interrupt Request** (HIRQ) - write-1-to-clear
  This bit is set when the host writes a HOSTDP IRQ control command to the configuration port. When set, this bit requests a HOSTDP status interrupt. The interrupt service routine (ISR) must write this bit to one to clear it.

- **Allow Configurations** (ALLOW_CNFG) - read-only
  The host processor is required to poll this bit to see when the Host DMA port is enabled and configuration writes are allowed. This bit is cleared when the last configuration word (YMODIFY) is written by the host. The bit is set again when the descriptor is completely passed to the DMA channel.
- **DMA Direction** (DMA_DIR) - read-only
  This bit is set to 0 when DMA is set for read and set to 1 for DMA writes. It reflects the WNR bit in the DMA_CONFIG word. If a former work unit was active, the bit does not update until the DMA_CPLT bit is set by software.
- **Bus Timeout Enabled** (BTE) - read-only
  This bit is just a copy of the BT_EN bit in the HOST_CONTROL register. The host can read this bit to determine if software has enabled the bus timeout feature.

This bit must be set by the interrupt service routine software which is called when the DMA is finished.

HOSTDP Timeout (HOST_TIMEOUT) Register

The HOSTDP timeout feature is previously described in "Acknowledge Mode" on page 8-10.

HOSTDP Timeout Register (HOST_TIMEOUT)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0xFFC03A08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset = 0x0000

Figure 8-13. HOSTDP Timeout Register

The HOSTDP timeout register (HOST_TIMEOUT), shown in Figure 8-13, holds the timeout value. A timer is loaded with this value when a host transaction is started. If HOSTDP does not respond with HRDY within the programmed amount of time, the TIMEOUT bit in the HOST_STATUS register is set and an interrupt is generated. This feature takes effect only when the BT_EN bit in the HOST_CONTROL register is set to 1.

The length of the timeout generated by this register is governed by the following equation:

$$timeout = (2\text{^}16 * HOST\_TIMEOUT)/(sclk\_freq)$$

For example, using an SCLK frequency of 133 MHz and HOST_TIMEOUT = 0x7ED, the timeout period is approximately one second.

Programming Examples

Listing 8-1. Enable 8-Bit HOSTDP data in pin MUXing

```
/* Enable 8-bit HOSTDP data in pin MUXing */

P5.H = hi(PORTD_FER);
P5.L = lo(PORTD_FER);
R5.L = PD15 | PD14 | PD13 | PD12 | PD11 | PD10 | PD9 | PD8 | nPD7
| nPD6 | nPD5 | nPD4 | nPD3 | nPD2 | nPD1 | nPD0;
w[P5] = R5.L;

P5.H=hi(PORTD_MUX);
P5.L=lo(PORTD_MUX);
R5.H=hi(MUX(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0));
R5.L=lo(MUX(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0));
[P5] = R5;
```

```
/* Enable 16-bit HOSTDP data in pin MUXing */

P5.H = hi(PORTD_FER);
P5.L = lo(PORTD_FER);
R5.L = PD15 | PD14 | PD13 | PD12 | PD11 | PD10 | PD9 | PD8 | PD7
| PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0;
w[P5] = R5.L;

P5.H=hi(PORTD_MUX);
P5.L=lo(PORTD_MUX);
R5.H=hi(MUX(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1));
R5.L=lo(MUX(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1));
[P5] = R5;
/* Enable HOSTDP Control Signals in pin MUXing */

P5.H = hi(PORTG_FER);
P5.L = lo(PORTG_FER);
R5.L = nPG15 | nPG14 | nPG13 | nPG12 | PG11 | nPG10 | nPG9 | nPG8
| PG7 | PG6 | PG5 | nPG4 | nPG3 | nPG2 | nPG1 | nPG0;
w[P5] = R5.L;

P5.H=hi(PORTH_MUX);
P5.L=lo(PORTH_MUX);
R5.H=hi(MUX(0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0));
R5.L=lo(MUX(0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0));
[P5] = R5;

P5.H = hi(PORTH_FER);
P5.L = lo(PORTH_FER);
R5.L = nPH15 | nPH14 | nPH13 | nPH12 | nPH11 | nPH10 | nPH9 | nPH8
| nPH7 | nPH6 | nPH5 | PH4 | PH3 | nPH2 | nPH1 | nPH0;
w[P5] = R5.L;
```

```
P5.H=hi(PORTH_MUX);
P5.L=lo(PORTH_MUX);
R5.H=hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));
R5.L=lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
[P5] = R5;
/* Enable 16-bit HOSTDP */

P5.H = hi(HOST_CONTROL);
P5.L = lo(HOST_CONTROL);
R5 = HOSTDP_EN  /* HOSTDP Enable */
| nHOSTDP_END /* Little endian transfers */
| HOSTDP_DATA_SIZE   /* 16-bit Data Size */
| nINT_MODE /* Acknowledge Mode */
| nBT_EN /* Bus timeout disabled */
| EHW /* Enable Host Writes */
| EHR /* Enable Host Reads */
| BDR (z); /* Burst DMA Requests On */
```

9 GENERAL-PURPOSE PORTS

This chapter describes general-purpose ports, pin multiplexing, general-purpose input/output (GPIO) functionality, and pin interrupts. This chapter includes the following sections:

- "Overview" on page 9-1
- "Module Overview" on page 9-3
- "Pin Multiplexing Scheme" on page 9-4
- "GPIO Functionality" on page 9-24
- "Pin Interrupts" on page 9-26
- "Programming Model" on page 9-29
- "Port Registers" on page 9-33
- "Programming Examples" on page 9-66

Overview

The general-purpose ports cover three jobs:

- Pin multiplexing scheme
- GPIO functionality
- Pin interrupts

This chapter characterizes each of the three topics in detail.

Features

The peripheral pins are functionally organized into ten general-purpose ports designated port A through port J. These ports feature:

- Up to 152 general-purpose I/O (GPIO) pins
- Input mode, output mode, and open-drain mode of GPIO operation
- Port multiplexing controlled by individual pin-per-pin base
- Identical port multiplexing scheme on all ADSP-BF54x processor Blackfin processor family derivatives
- No glue hardware required for unused pins
- Four interrupt channels dedicated to pin interrupts
- All port pins provide interrupt functionality
- Byte-wide pin-to-interrupt assignment

Module Overview

A simplified illustration of the GPIO and pin interrupt signal flow is shown in Figure 9-1.

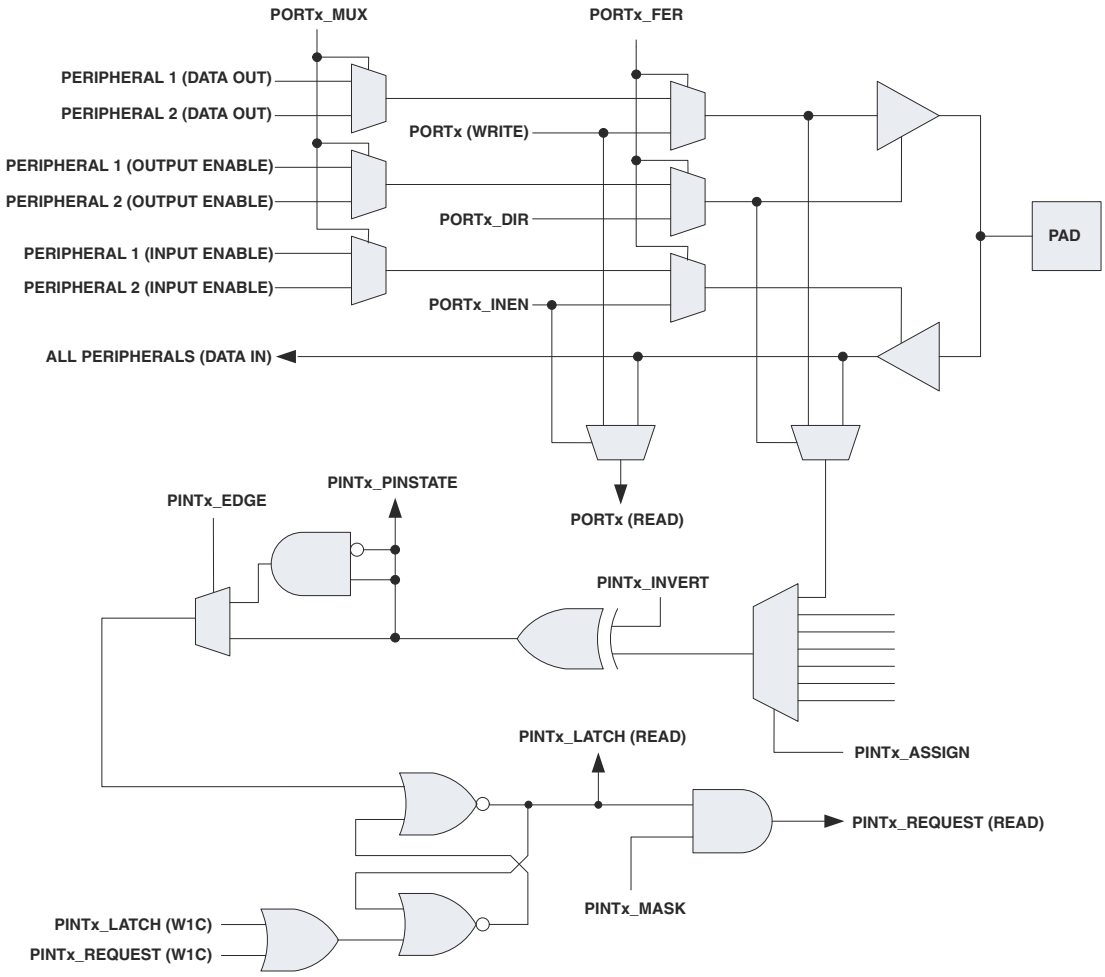


Figure 9-1. Simplified GPIO and Pin Interrupt Signal Flow

External Interfaces

The pin multiplexing hardware can be seen as a layer between the on-chip peripherals and the pads of the silicon. All pins grouped into the ports “port A” to “port J” are controlled by this unit.

Internal Interfaces

All MMR registers of the pin multiplexing, GPIO and pin interrupt control blocks can be accessed through the PAB bus. There is no DMA support. Every one of the four pin interrupt modules has its own and dedicated interrupt request output signal that connects directly to the SIC controller, as shown in Figure 9-2 on page 9-26.

Pin Multiplexing Scheme

ADSP-BF54x processor Blackfin processors feature a rich set of on-chip peripherals. Each set of peripherals has a combination of input and output signals associated with them. In total, these are many more signals than pins available on the processors. Therefore, a powerful pin multiplexing scheme provides best flexibility to external application space.

Table 9-1 shows all peripheral signals that are accessible off the chip through the general-purpose ports. The individual members of the ADSP-BF54x processor Blackfin processor family do not feature all the listed peripherals at the same time. Note that some signals are optional and are not necessarily required in all operating modes.

Table 9-1. General-Purpose and Special Function Signals

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
EBIU (async)	Address (22)	H, I	x	x	x	x	x
	Bus Handshake (3)	J					
	Clock (1)	I					
	Ready (1)	J					
NAND Flash Controller	Control (2)	J	x	x	x	x	x
ATAPI	Control (8)	J	x	x	x	-	x
	Reset (1)	H					
HostDMA Port (HOSTDP)	Data (16)	D	x	x	x	x	-
	Control (3)	B, G, H					
	Address (1)	H					
	Acknowledge (1)	H					
SD/SDIO Controller	Data (4)	C	x	x	x	-	x
	Clock (1)	C					
	Command (1)	C					
EPPI0	Data (24)	D, F	x	x	x	x	-
	Clock (1)	G					
	Frame Sync (3)	G, H					
EPPI1	Data (16)	D	x	x	x	x	x
	Clock (1)	E					
	Frame Sync (3)	E, H					
EPPI2	Data (8)	D	x	x	x	x	x
	Clock (1)	G					
	Frame Sync (3)	G, H					

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
SPORT0	Data (4)	C	x	x	x	-	-
	Clock (2)	C					
	Frame Sync (2)	C					
SPORT1	Data (4)	D	x	x	x	x	x
	Clock (2)	D					
	Frame Sync (2)	D					
SPORT2	Data (4)	A	x	x	x	x	x
	Clock (2)	A					
	Frame Sync (2)	A					
SPORT3	Data (4)	A	x	x	x	x	x
	Clock (2)	A					
	Frame Sync (2)	A					
SPI0	Data (2)	E	x	x	x	x	x
	Clock (1)	E					
	Slave Select (1)	E					
	Slave Enable (3)	E					
SPI1	Data (2)	G	x	x	x	x	x
	Clock (1)	G					
	Slave Select (1)	G					
	Slave Enable (3)	G					
SPI2	Data (2)	B	–x	–x	x	-	-
	Clock (1)	B					
	Slave Select (1)	B					
	Slave Enable (3)	B					

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
UART0	Data (2)	E	x	x	x	x	x
UART1	Data (2)	H	x	x	x	x	x
	Control (2)	E					
UART2	Data (2)	B	x	x	x	-	-
UART3	Data (2)	B	x	x	x	x	x
	Control (2)	B					
High Speed USB OTG			x	x	x	-	x
CAN0[1]	Data (2)	G	x	x	-	x	x
CAN1[1]	Data (2)	G	–	x	-	x	-
TWI0	Data (1)	E	x	x	x	x	x
	Clock (1)	E					
TWI1	Data (1)	B	x	x	x	x	-
	Clock (1)	B					
Timer 0-7	PWM/Capture/Clock (8)	A, B	x	x	x	x	x
	Alternate Clock Input (8)	A					
	Alternate Capture Input (7)	A, B, E, G, H					
Timer 8-10	PWM/Capture/Clock (3)	H	x	x	x	x	-
	Alternate Clock Input (3)	H					
	Alternate Capture Input (3)	H					
Up/ Down Counter	Up / Dir (1)	H	x	x	x	x	x
	Down / Gate (1)	H					
	Zero Marker (1)	G					

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
KEYPAD	Rows (8)	D, E	x	–x	x	-	x
	Columns (8)	D, E					
MXVR	Data (2)	H	x	–	–	-	-
	Clock (2)	C					
	Control (2)	G, H					
GPIOs	GPIOs (152)	A-J	x	x	x	x	x

1 Automotive only.

Read from Page 0x05 of the on-chip OTP memory when determining whether a module is available on a respective ADSP-BF54x processor Blackfin processor. For details, see "System Reset and Booting" on page 17-1.

The peripheral pins of the ADSP-BF54x processor Blackfin processors are functionally organized into ten general-purpose ports which are designated Port A through port J. Most ports consist of 16 pins; a few have fewer. By default, all port pins are configured for GPIO operation after reset. In total, there are 152 GPIO-capable pins. Pin interrupt functionality is covered by a separate functional block.

The individual ports are discussed in the following sections.

Port A

Port A consists of 16 pins, referred to as PA0 to PA15, as shown in Table 9-2. Besides the 16 GPIOs, this port homes all SPORT2 and SPORT3 signals. If the secondary data pins are not needed, the corresponding pins can be used for general-purpose timer purposes.

Table 9-2. Port A Pin Configuration

Pin	GPIO	PORTA_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PA0	PA0	1:0	SPORT2 TFS	-	-	-	-
PA1	PA1	3:2	SPORT2 DTSEC	TMR4	-	-	-
PA2	PA2	5:4	SPORT2 DTPRI	-	-	-	-
PA3	PA3	7:6	SPORT2 TSCLK	-	-	-	-
PA4	PA4	9:8	SPORT2 RFS	-	-	-	-
PA5	PA5	11:10	SPORT2 DRSEC	TMR5	-	-	-
PA6	PA6	13:12	SPORT2 DRPRI	-	-	-	-
PA7	PA7	15:14	SPORT2 RSCLK	-	-	-	TACLK0 [1]
PA8	PA8	17:16	SPORT3 TFS	-	-	-	TACLK1 [1]
PA9	PA9	19:18	SPORT3 DTSEC	TMR6	-	-	-
PA10	PA10	21:20	SPORT3 DTPRI	-	-	-	TACLK2 [1]
PA11	PA11	23:22	SPORT3 TSCLK	-	-	-	TACLK3 [1]
PA12	PA12	25:24	SPORT3 RFS	-	-	-	TACLK4 [1]

Table 9-2. Port A Pin Configuration (Cont'd)

Pin	GPIO	PORTA_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PA13	PA13	27:26	SPORT3 DRSEC	TMR7	-	-	TACLK5[1]
PA14	PA14	29:28	SPORT3 DRPRI	-	-	-	TACLK6[1]
PA15	PA15	31:30	SPORT3 RSCLK	-	-	-	TACI7[1], TACLK7[1]

1 To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Port B

Port B consists of 15 pins, referred to as `PB0` to `PB14`, as shown in Table 9-3. Besides the 15 GPIOs, this port homes `TW1`, `UART2`, `UART3` and `SPI2` signals. If the `SPI2` slave select signals are not needed, the corresponding pins can be used for general-purpose timer purposes.

Table 9-3. Port B Pin Configuration

Pin	GPIO	PORTB_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PB0	PB0	1:0	TWI1 SCL[1]	-	-	-	-
PB1	PB1	3:2	TWI1 SDA[1]	-	-	-	-
PB2	PB2	5:4	UART3 RTS	-	-	-	-
PB3	PB3	7:6	UART3 CTS	-	-	-	-
PB4	PB4	9:8	UART2 TX	-	-	-	-

Table 9-3. Port B Pin Configuration (Cont'd)

Pin	GPIO	PORTB_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PB5	PB5	11:10	UART2 RX	-	-	-	TACI2[2]
PB6	PB6	13:12	UART3 TX	-	-	-	-
PB7	PB7	15:14	UART3 RX	-	-	-	TACI3[2]
PB8	PB8	17:16	SPI2 SS	TMR0	-	-	-
PB9	PB9	19:18	SPI2 SSEL1	TMR1	-	-	-
PB10	PB10	21:20	SPI2 SSEL2	TMR2	-	-	-
PB11	PB11	23:22	SPI2 SSEL3	TMR3	-	-	HWAIT[3]
PB12	PB12	25:24	SPI2 SCK	-	-	-	-
PB13	PB13	27:26	SPI2 MOSI	-	-	-	-
PB14	PB14	29:28	SPI2 MISO	-	-	-	-

1 PB_0 and PB_1 are I^2C pins which also have GPIO capability. Since the I^2C pads can only drive low, the GPIO for these two bits cannot drive a 1. These pads should be used with an external pull-up, so that a 1 is seen when they are not pulling down.

2 To enable timer alternate capture and clock Inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

3 The Boot Host Wait (HWAIT) signal is a GPIO output that is driven and toggled by the boot kernel at boot time. An external pulling resistor is required for proper operation. A pull-up resistor instructs the HWAIT signal to behave active high (low when ready for data). A pull-down resistor instructs the HWAIT signal to behave active low (high when ready for data). After boot it can be used for other purposes. If PB11 is used for other purposes (for example, timer or SPI operation), the HWAITA signal on PH7 can be used alternatively. The Alternate Host Wait (HWAITA) can be alternatively used instead of HWAIT on PH7 when programming the OTP_ALTERNATE_HWAIT bit in the PBS_MAIN_LO OTP memory page. For details, see "System Reset and Booting" on page 17-1.

Port C

Port C consists of 14 pins, referred to as PC0 to PC13, as shown in Table 9-4. Besides the 14 GPIOs, this port homes SPORT0 and SDIO signals.

Table 9-4. Port C Pin Configuration

Pin	GPIO	PORTC_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PC0	PC0	1:0	SPORT0 TFS	-	-	-	-
PC1	PC1	3:2	SPORT0 DTSEC	MMCLK	-	-	-
PC2	PC2	5:4	SPORT0 DTPRI	-	-	-	-
PC3	PC3	7:6	SPORT0 TSCLK	-	-	-	-
PC4	PC4	9:8	SPORT0 RFS	-	-	-	-
PC5	PC5	11:10	SPORT0 DRSEC	MBCLK	-	-	-
PC6	PC6	13:12	SPORT0 DRPRI	-	-	-	-
PC7	PC7	15:14	SPORT0 RSCLK	-	-	-	-
PC8	PC8	17:16	SD D0	-	-	-	-
PC9	PC9	19:18	SD D1	-	-	-	-
PC10	PC10	21:20	SD D2	-	-	-	-
PC11	PC11	23:22	SD D3	-	-	-	-
PC12	PC12	25:24	SD CLK	-	-	-	-
PC13	PC13	27:26	SD CMD	-	-	-	-

Port D

Port D consists of 16 pins, referred to as PD0 to PD15, as shown in Table 9-5. Besides the 16 GPIOs, this port homes data signals of all three EPPI ports and of the host port. Additionally, there are the SPORT1 signals and four columns and four rows of the keypad peripheral.

This port provides flexible configurations, whereby 8-, 16-, or 24-bit EPPI configurations can be balanced against 8- or 16-bit host operation.

Table 9-5. Port D Pin Configuration

Pin	GPIO	PORTD_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PD0	PD0	1:0	PPI1 D0	HOST D8	SPORT1 TFS	PPI0 D18	-
PD1	PD1	3:2	PPI1 D1	HOST D9	SPORT1 DTSEC	PPI0 D19	-
PD2	PD2	5:4	PPI1 D2	HOST D10	SPORT1 DTPRI	PPI0 D20	-
PD3	PD3	7:6	PPI1 D3	HOST D11	SPORT1 TSCLK	PPI0 D21	-
PD4	PD4	9:8	PPI1 D4	HOST D12	SPORT1 RFS	PPI0 D22	-
PD5	PD5	11:10	PPI1 D5	HOST D13	SPORT1 DRSEC	PPI0 D23	-
PD6	PD6	13:12	PPI1 D6	HOST D14	SPORT1 DRPRI	-	-
PD7	PD7	15:14	PPI1 D7	HOST D15	SPORT1 RSCLK	-	-
PD8	PD8	17:16	PPI1 D8	HOST D0	PPI2 D0	KEY ROW0	-
PD9	PD9	19:18	PPI1 D9	HOST D1	PPI2 D1	KEY ROW1	-
PD10	PD10	21:20	PPI1 D10	HOST D2	PPI2 D2	KEY ROW2	-
PD11	PD11	23:22	PPI1 D11	HOST D3	PPI2 D3	KEY ROW3	-

Table 9-5. Port D Pin Configuration (Cont'd)

Pin	GPIO	PORTD_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PD12	PD12	25:24	PPI1 D12	HOST D4	PPI2 D4	KEY COL0	-
PD13	PD13	27:26	PPI1 D13	HOST D5	PPI2 D5	KEY COL1	-
PD14	PD14	29:28	PPI1 D14	HOST D6	PPI2 D6	KEY COL2	-
PD15	PD15	31:30	PPI1 D15	HOST D7	PPI2 D7	KEY COL3	-

Port E

Port E consists of 16 pins, referred to as PE0 to PE15, as shown in Table 9-6. Besides the 16 GPIOs, this port homes data signals for SPI0, UART0, and TWI0. Furthermore, there are UART1 hardware flow control signals and PPI1 clock and frame sync signals. If not all signals of the SPI0 are needed in an application, the associated pins can operate as rows and columns for the keypad peripheral.

Table 9-6. Port E Pin Configuration

Pin	GPIO	PORTE_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PE0	PE0	1:0	SPI0 SCK	KEY COL7	-	-	-
PE1	PE1	3:2	SPI0 MISO	KEY ROW6	-	-	-
PE2	PE2	5:4	SPI0 MOSI	KEY COL6	-	-	-
PE3	PE3	7:6	SPI0 SS	KEY ROW5	-	-	-
PE4	PE4	9:8	SPI0 SEL1	KEY COL5	-	-	-
PE5	PE5	11:10	SPI0 SEL2	KEY ROW4	-	-	-

Table 9-6. Port E Pin Configuration (Cont'd)

Pin	GPIO	PORTE_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PE6	PE6	13:12	SPI0 SEL3	KEY COL4	-	-	-
PE7	PE7	15:14	UART0 TX	KEY ROW7	-	-	-
PE8	PE8	17:16	UART0 RX	-	-	-	TACI0[1]
PE9	PE9	19:18	UART1 RTS	-	-	-	-
PE10	PE10	21:20	UART1 CTS	-	-	-	-
PE11	PE11	23:22	PPI1 CLK	-	-	-	-
PE12	PE12	25:24	PPI1 FS1	-	-	-	-
PE13	PE13	27:26	PPI1 FS2	-	-	-	-
PE14	PE14	29:28	TWI0 SCL	-	-	-	-
PE15	PE15	31:30	TWI0 SDA	-	-	-	-

1 To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Port F

Port F consists of 16 pins, referred to as `PF0` to `PF15`, as shown in Table 9-7. Besides the 16 GPIOs, this port homes 16 data signals of the PPI0 interface. This port can alternatively provide the ATAPI data signals if not multiplexed with the asynchronous bus.

Table 9-7. Port F Pin Configuration

Pin	GPIO	PORTF_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PF0	PF0	1:0	PPI0 D0	ATAPI D0A[1]	-	-	-
PF1	PF1	3:2	PPI0 D1	ATAPI D1A[1]	-	-	-
PF2	PF2	5:4	PPI0 D2	ATAPI D2A[1]	-	-	-
PF3	PF3	7:6	PPI0 D3	ATAPI D3A[1]	-	-	-
PF4	PF4	9:8	PPI0 D4	ATAPI D4A[1]	-	-	-
PF5	PF5	11:10	PPI0 D5	ATAPI D5A[1]	-	-	-
PF6	PF6	13:12	PPI0 D6	ATAPI D6A[1]	-	-	-
PF7	PF7	15:14	PPI0 D7	ATAPI D7A[1]	-	-	-
PF8	PF8	17:16	PPI0 D8	ATAPI D8A[1]	-	-	-
PF9	PF9	19:18	PPI0 D9	ATAPI D9A[1]	-	-	-
PF10	PF10	21:20	PPI0 D10	ATAPI D10A[1]	-	-	-
PF11	PF11	23:22	PPI0 D11	ATAPI D11A[1]	-	-	-
PF12	PF12	25:24	PPI0 D12	ATAPI D12A[1]	-	-	-
PF13	PF13	27:26	PPI0 D13	ATAPI D13A[1]	-	-	-
PF14	PF14	29:28	PPI0 D14	ATAPI D14A[1]	-	-	-
PF15	PF15	31:30	PPI0 D15	ATAPI D15A[1]	-	-	-

1 ATAPI data and address signals are routed to alternate homes when PORTF_MUX[1:0] == b#01.

Port G

Port G consists of 16 pins, referred to as PG0 to PG15, as shown in Table 9-8. Besides the 16 GPIOs, this port homes EPPI0 control signals, all CAN signals, as well as the SPI1 signals. If additional SPI1 slave select signals are not needed by an application, the associated pins can alternatively function as Host DMA port or EPPI2 control signals. Also, the zero marker input of the counter module is there.

Table 9-8. Port G Pin Configuration

Pin	GPIO	PORTG_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PG0	PG0	1:0	PPI0 CLK	-	-	-	TMRCLK[1]
PG1	PG1	3:2	PPI0 FS1	-	-	-	-
PG2	PG2	5:4	PPI0 FS2	ATAPI A0A[2]	-	-	-
PG3	PG3	7:6	PPI0 D16	ATAPI A1A[2]	-	-	-
PG4	PG4	9:8	PPI0 D17	ATAPI A2A[2]	-	-	-
PG5	PG5	11:10	SPI1 SEL1	HOST CE	PPI2 FS2	CNT CZM	-
PG6	PG6	13:12	SPI1 SEL2	HOST RD	PPI2 FS1	-	-
PG7	PG7	15:14	SPI1 SEL3	HOST WR	PPI2 CLK	-	-
PG8	PG8	17:16	SPI1 SCK	-	-	-	-
PG9	PG9	19:18	SPI1 MISO	-	-	-	-
PG10	PG10	21:20	SPI1 MOSI	-	-	-	-
PG11	PG11	23:22	SPI1 SS	MTXONB	-	-	-
PG12	PG12	25:24	CAN0 TX	-	-	-	-

Table 9-8. Port G Pin Configuration (Cont'd)

Pin	GPIO	PORTG_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PG13	PG13	27:26	CAN0 RX	-	-	-	TACI4[3]
PG14	PG14	29:28	CAN1 TX	-	-	-	-
PG15	PG15	31:30	CAN1 RX	-	-	-	TACI5[3]

1 TMRCLK serves all eleven general-purpose timers.
2 ATAPI data and address signals are routed to alternate homes when PORTF_MUX[1:0] == b#01.
3 To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Port H

Port H consists of 14 pins, referred to as PH0 to PH13, as shown in Table 9-9. Besides the 14 GPIOs, this port homes six address lines of the parallel asynchronous memory interface. Furthermore, there are the UART1 data signals and set of miscellaneous control signals, such as Host DMA port strobes, handshaked-memory DMA request strobes, the third EPPI frame syncs, and the up- and down-count inputs of the counter module.

The boot host wait (HWAIT) and alternate boot host wait (HWAITA) are not associated with any hardware block. It is a normal GPIO pin that has a special purpose during booting. For details, see "System Reset and Booting" on page 17-1.

Table 9-9. Port H Pin Configuration

Pin	GPIO	PORTH_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PH0	PH0	1:0	UART1 TX	PPI1 FS3	-	-	-
PH1	PH1	3:2	UART1 RX	EPPI0 FS3	-	-	TACI1[1]
PH2	PH2	5:4	ATAPI RESET	TMR8	EPPI2 FS3	-	-
PH3	PH3	7:6	HOST ADDR	TMR9	CNT CDG	-	-
PH4	PH4	9:8	HOST ACK	TMR10	CNT CUD	-	-
PH5	PH5	11:10	MTX	DMAR0	-	-	TACI8[1], TACLK8[1]
PH6	PH6	13:12	MRX	DMAR1	-	-	TACI9[1], TACLK9[1]
PH7	PH7	15:14	MRXONB	-	-	-	TACI10[1], TACLK10[1], HWAIT[2]
PH8	PH8	17:16	A4	-	-	-	
PH9	PH9	19:18	A5	-	-	-	-
PH10	PH10	21:20	A6	-	-	-	-
PH11	PH11	23:22	A7	-	-	-	-
PH12	PH12	25:24	A8	-	-	-	-
PH13	PH13	27:26	A9	-	-	-	-

1 To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

2 The Boot Host Wait (HWAIT) signal is a GPIO output that is driven and toggled by the boot kernel at boot time. An external pulling resistor is required for proper operation. A pull-up resistor instructs the HWAIT signal to behave active high (low when ready for data). A pull-down resistor instructs the HWAIT signal to behave active low (high when ready for data). After boot, it can be used for other purposes. If PH7 is used for other purposes (for example, MXVR operation), the HWAITA signal on PB11 can be used alternatively. HWAITA operation is enabled by programming the OTP_ALTERNATE_HWAIT bit in the PBS_MAIN_LO OTP memory page. For details, see "System Reset and Booting" on page 17-1.

Port I

Port I consists of 16 pins, referred to as `PI0` to `PI15`, as shown in Table 9-10. Besides the 16 GPIOs, this port homes the upper 16 address lines of the parallel asynchronous memory interface and the clock for the synchronous NOR flash interface.

Table 9-10. Port I Pin Configuration

Pin	GPIO	PORTI_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PI0	PI0	1:0	A10	-	-	-	-
PI1	PI1	3:2	A11	-	-	-	-
PI2	PI2	5:4	A12	-	-	-	-
PI3	PI3	7:6	A13	-	-	-	-
PI4	PI4	9:8	A14	-	-	-	-
PI5	PI5	11:10	A15	-	-	-	-
PI6	PI6	13:12	A16	-	-	-	-
PI7	PI7	15:14	A17	-	-	-	-
PI8	PI8	17:16	A18	-	-	-	-
PI9	PI9	19:18	A19	-	-	-	-
PI10	PI10	21:20	A20	-	-	-	-

Table 9-10. Port I Pin Configuration (Cont'd)

Pin	GPIO	PORTI_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PI11	PI11	23:22	A21	-	-	-	-
PI12	PI12	25:24	A22	-	-	-	-
PI13	PI13	27:26	A23	-	-	-	-
PI14	PI14	29:28	A24	-	-	-	-
PI15	PI15	31:30	A25	NOR CLK	-	-	-

Port J

Port J consists of 16 pins, referred to as `PJ0` to `PJ15`, as shown in Table 9-11. Besides the 16 GPIOs, this port provides various control signals for the NAND flash, NOR flash, and ATAPI interfaces.

Table 9-11. Port J Pin Configuration

Pin	GPIO	PORTJ_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PJ0	PJ0	1:0	AMC ARDY / NOR WAIT	-	-	-	-
PJ1	PJ1	3:2	NAND CE	-	-	-	-
PJ2	PJ2	5:4	NAND RB	-	-	-	-
PJ3	PJ3	7:6	ATAPI DIOR	-	-	-	-
PJ4	PJ4	9:8	ATAPI DIOW	-	-	-	-
PJ5	PJ5	11:10	ATAPI CS0	-	-	-	-

Table 9-11. Port J Pin Configuration (Cont'd)

Pin	GPIO	PORTJ_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PJ6	PJ6	13:12	ATAPI CS1	-	-	-	-
PJ7	PJ7	15:14	ATAPI DMACK	-	-	-	-
PJ8	PJ8	17:16	ATAPI DMARQ	-	-	-	-
PJ9	PJ9	19:18	ATAPI INTRQ	-	-	-	-
PJ10	PJ10	21:20	ATAPI IORDY	-	-	-	-
PJ11	PJ11	23:22	AMC BR	-	-	-	-
PJ12	PJ12	25:24	AMC BG	-	-	-	-
PJ13	PJ13	27:26	AMC BGH	-	-	-	-

Port Multiplexing Control

By default, after reset, all port pins are in GPIO input mode with their output and input drivers disabled. As a result, all unused port pins can be left unconnected. Disabled pins appear in high-impedance mode to external circuits and are pulled low to internal logic.

Each port has two dedicated MMRs that control the port multiplexing, the 16-bit function enable (PORTx_FER) registers, and the 32-bit port multiplexing (PORTx_MUX) registers.

In this chapter, the naming convention for registers and bits uses a lowercase "x" to represent A to J. For example, the name PORTx_FER represents PORTA_FER, PORTB_FER, and so on, through PORTJ_FER.

The bit name PxO represents PA0, PB0, and so on, through PJ0. This convention is used in register descriptions common to the ten ports.

Each bit in the 16-bit PORTx_FER registers represents one port pin. For example, bit 1 of the PORTA_FER register sets the PA1 pin to GPIO operation mode when cleared. When set, one of the available peripheral functions becomes active. The PA1 pin can either operate as a secondary transmit data signal of SPORT2 or as PWM/capture/clock pin of Timer 4.

Every pair of bits in the PORTx_MUX registers controls the multiplexing between the peripheral functions available to a pin. This is a 2-bit field because some pins provide up to four options. The truth table of the bit field is identical to all ADSP-BF54x processor Blackfin processor family derivatives, regardless of whether all options are available on a given silicon.

In the case of the PA1 example, bit 3 and bit 2 control the multiplexer of the PA1 pin. The truth table of the entire function enable and multiplexing control is shown in Table 9-12.

Table 9-12. Port Multiplexing Control Example

PORTA_FER [1]	PORTA_MUX [3:2]	PA1 Function
0	00	GPIO
0	01	GPIO
0	10	GPIO
0	11	GPIO
1	00	SPORT2 DTSEC
1	01	TMR4
1	10	Reserved
1	11	Reserved

The port multiplexing scheme provides best granularity, as every pin can be controlled on an individual basis. If SPORT2 is used in any mode that does not require the secondary transmit data feature, the PA1 pin can still be used as GPIO or as TMR4.

GPIO Functionality

Every port pin can operate in GPIO mode. This is the default after reset and is controlled by the port-specific PORTx_FER function enable register. Every port has a dedicated set of MMR registers that control GPIO functionality. Every bit in these registers represents a certain GPIO pin of the specific port. Refer to Figure 9-2 for a related diagram.

In this chapter, the naming convention for registers and bits uses a lowercase "x" to represent A through J. For example, the name PORTx_FER represents PORTA_FER, PORTB_FER, and so on, through PORTJ_FER. The bit name PxO represents PA0, PB0, and so on, through PJ0. This convention is used to discuss registers common to the ten ports.

By default, every GPIO is in input mode. The input drivers are not enabled which avoids the need for unnecessary current sinks and the external pulling of resistors on unused or do not care pins.

Input Mode

The default mode of every GPIO pin after reset is input mode, but the input drivers are not enabled. To enable any GPIO input drivers, set the corresponding bits in the input enable register PORTx_INEN. When enabled, a read from the PORTx register returns the logical state of the input pin. The input signal does not overwrite the state of the flip-flop used for the output case. That state can only be altered by software. If the input driver is enabled, a write to the PORTx register can alter the state of the flip-flop, but the change cannot be read back.

Output Mode

Any GPIO pin can be configured for output mode. The GPIO output drivers are enabled by setting the corresponding bits in the direction registers. Direction registers are implemented as a pair of write-1-to-set (W1S) and write-1-to-clear (W1C) MMRs, called PORTx_DIR_SET and PORTx_DIR_CLEAR. This way, direction of the signal flow of individual GPIO pins can be altered by separate software threads without mutually impacting other GPIOs on the same port. Both registers return the same value when read. A logical 1 indicates an enabled output.

The state of output pins is controlled by the PORTx registers. A logical 0 drives the output low. A logical 1 drives the output high. While the PORTx register can be written to alter all GPIOs of a specific port at once, there is also a pair of W1S and W1C MMRs, called PORTx_SET and PORTx_CLEAR that enable manipulation of individual GPIO outputs. The state of the outputs can be obtained by reading the PORTx registers.

Because the state of the GPIO output can already be controlled before the output driver is enabled, it is recommended to first set or clear the flip-flop to avoid any volatile levels on the output.

Open-Drain Mode

Every GPIO can also be used in open-drain mode. To accomplish this, first, clear the respective bit in the PORTx or PORTx_CLEAR register then set the one bit in the PORTx_INEN register. Reads from the PORTx register then return the status from the pin and do not return the state of the internal flip-flop. By toggling the output driver through the PORTx_DIR_SET and PORTx_DIR_CLEAR register pair, the output signal can be pulled low or three-stated as required. Note that the polarity of the driven signal can be inverted when the internal flip-flop is set instead. When a GPIO port is used in open-drain mode, care must be taken not to exceed the VIH operating condition associated with the respective pin.

Pin Interrupts

On the ADSP-BF54x processor Blackfin processor family, the pin interrupts have been completely decoupled from basic GPIO functionality due to the following set of advantages:

- Flexible mapping scheme enables pins from up to four different ports to be grouped to one common interrupt scheme.
- Interrupts work on input and output pins regardless of whether in GPIO or functional mode.

ADSP-BF54x processor Blackfin processors have four SIC interrupt channels dedicated to pin interrupt purposes. These channels are managed by four hardware blocks, called `PINT0`, `PINT1`, `PINT2`, and `PINT3`. Every `PINTx` block can sense to up to 32 pins. While `PINT0` and `PINT1` can sense the pins of port A and port B, `PINT2` and `PINT3` manage all the pins from port C to port J as shown in Figure 9-2.

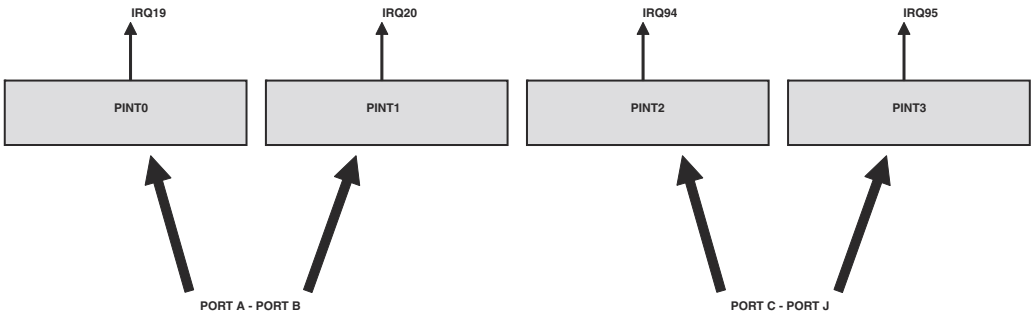


Figure 9-2. Signal Flow

The diagram shown in Figure 9-1 on page 9-3 shows the signal flow from the pin through the `PINTx` module to the SIC controller. Special attention is required with regard to how the pins are assigned to the `PINTx` modules as shown in Figure 9-3.

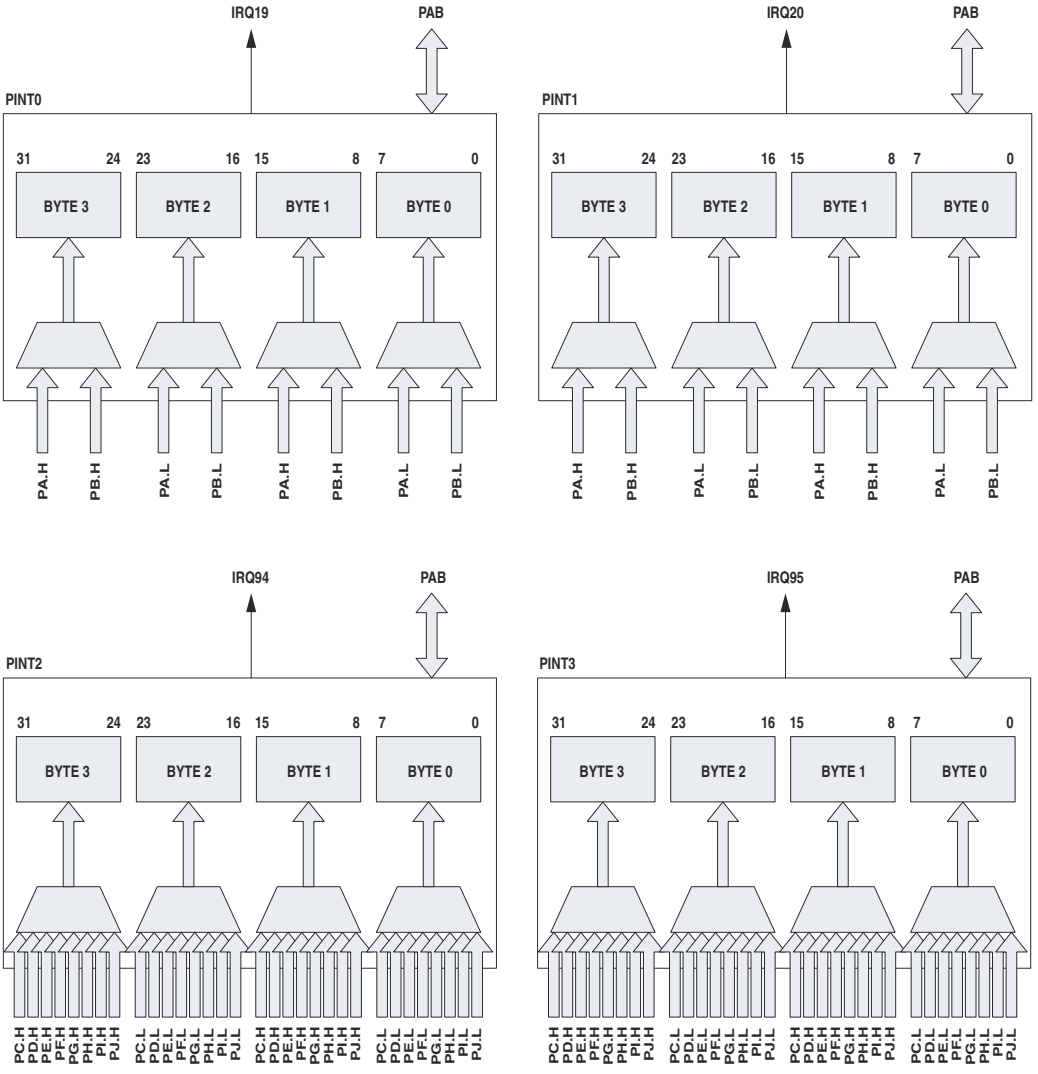


Figure 9-3. Pin-to-Interrupt Assignment

The ten ports are subdivided into 8-bit half ports, resulting in lower and upper half 8-bit units. The `PINTx_ASSIGN` registers control the 8-bit multiplexers shown in Figure 9-3. Lower half units of eight pins can be

forwarded to either byte 0 or byte 2 of either associated PINTx block. Upper half units can be forwarded to either byte 1 or byte 3 of the pin interrupt blocks, without further restrictions.

When a half port is assigned to a byte in any PINTx block, the state of the eight pins (regardless of GPIO or function, input or output) can be seen in the PINTx_PINSTATE register. While neither input nor output drivers of the pin are enabled, the pin state is read as zero. The PINTx_PINSTATE register reports the inverted state of the pin if the signal inverter is activated by the PINTx_INVERT_SET register. The inverter can be enabled on a individual bit by bit basis. Every bit in the PINTx_INVERT_SET/CLEAR register pair represents a pin signal.

As shown in Figure 9-1 on page 9-3, the interrupt can be generated on an active high level of the signal or a raising edge of the signal. The default behavior is level sensitivity. PINTx_EDGE_SET register can be used to change the behavior to edge sensitivity. By enabling the inverter using the PINTx_INVERT_SET register, the interrupt behavior can be altered to trigger on active-low signals or falling edges.

The PINTx modules also assist if both signal edges are required to generate interrupts. If two different interrupt requests are required, the PINTx_ASSIGN registers can route a signal to two different PINTx blocks, where one block inverts the signal and the other one does not. If both signal edges can report over the same interrupt, every signal can be routed through to different bit positions within a single PINTx block, where the inverted should be enabled for either one. The servicing software routine can then tell from the PINTx_LATCH whether a falling, a rising or both edges have occurred.

Regardless whether in level-sensitive or edge-sensitive mode, an interrupt is always latched by the hardware. Latched signals can be read from the PINTx_LATCH registers. Latches can only be cleared by software or a hardware reset. To clear, W1C the PINTx_REQUEST or the PINTx_LATCH register. If the pin state does not change by the time the interrupt service routine returns, the interrupt is requested again, when in level-sensitive mode.

Because every PINTx block groups up to 32 pin signals, the PINTx_MASK_SET/CLEAR register pair can control which of the signals can request an interrupt at system level. Software may interrogate the PINTx_REQUEST register for signaling pins. PINTx_REQUEST bits represent a logical AND between the mask and the latch. When any of these bits is set, an interrupt is forwarded to the SIC controller.

All MMR registers in the pin interrupt module are 32 bits wide. Individual bits of PINTx registers represent the associated pins. Nevertheless, the 32 bits can also be seen as four groups of eight bits. Each group can manage up to eight pins out of either the lower or an upper half of any associated port.

Programming Model

Figure 9-4, Figure 9-5, and Figure 9-6 show the programming model of the general-purpose ports. This includes GPIO input and output operation, as well as open-drain mode. Figure 9-6 (the third part of the diagram) illustrates the model of the pin interrupt PINTx modules.

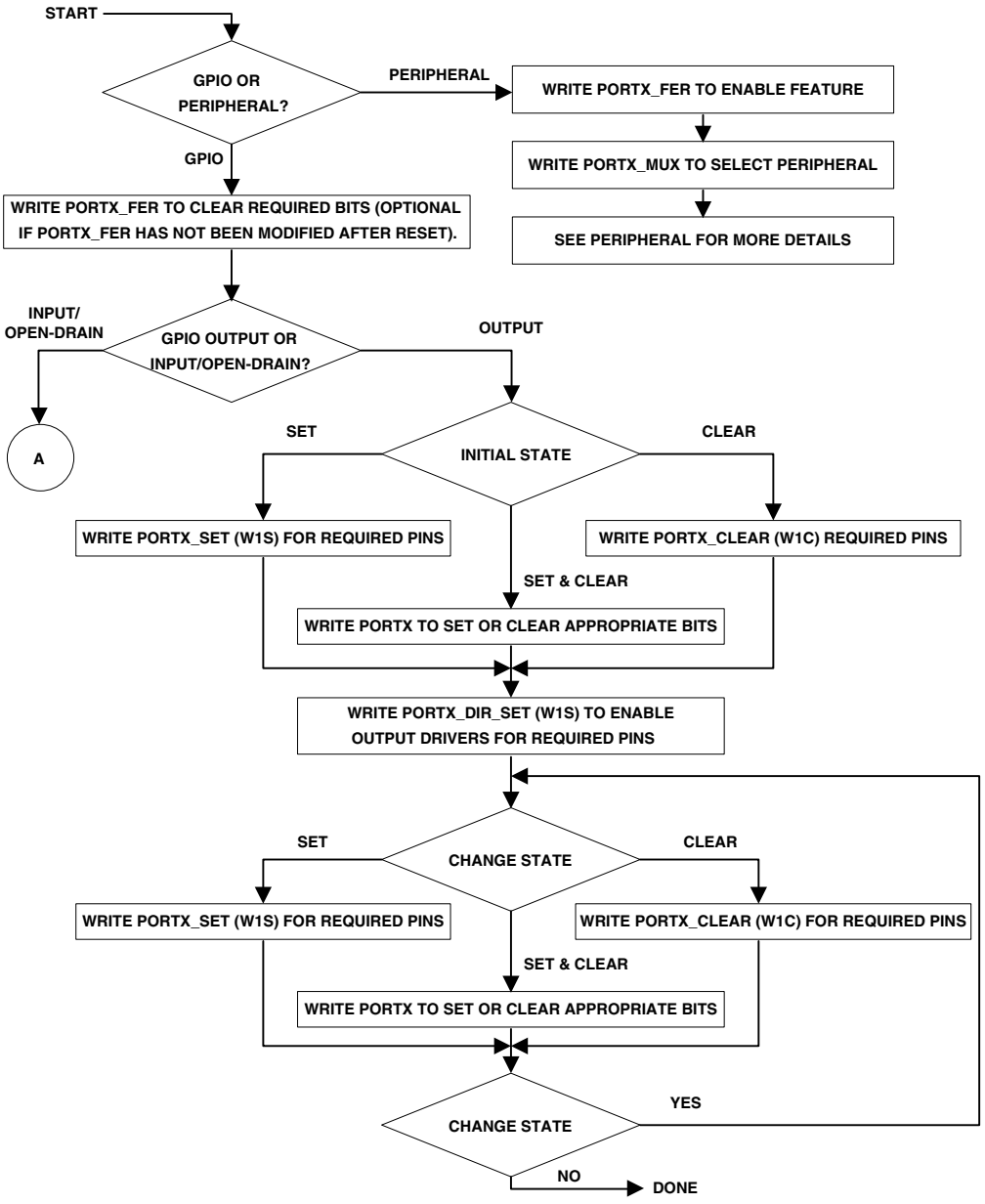
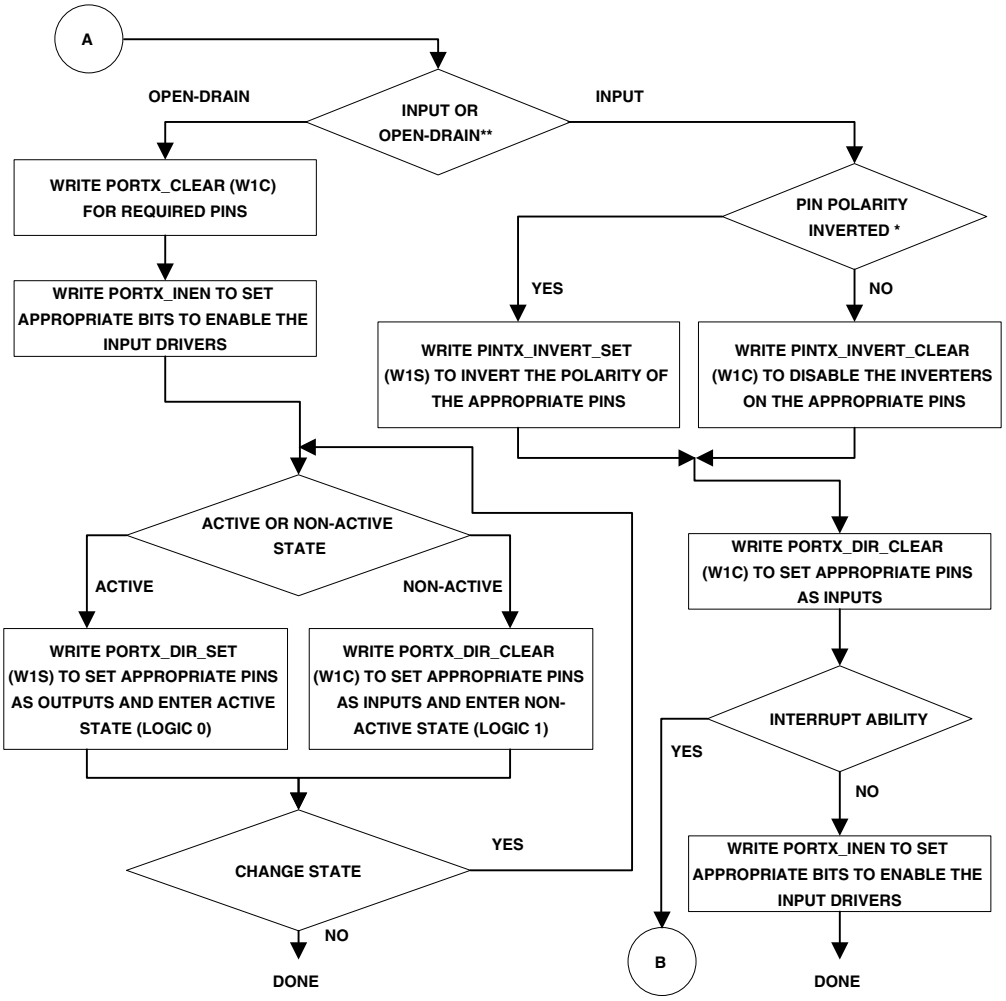


Figure 9-4. GPIO Programming Model Flow (Part 1)



*** The pin polarity set at this point will effect the behaviour of the interrupt functionality detailed in the next figure. If the invert bit is set for a given pin, and edge sensitive interrupts are configured. The interrupt will be latched on detection of a falling edge. If the inverse bit is clear, edge sensitive interrupts are generated on the rising edge. For level sensitive interrupts, enabling the inverter will result in interrupts being detected on a low signal. Disabling the inverter will result in interrupts being latched on high signals.**

**** Open-drain mode assumes an external pull-up resistor is fitted**

Figure 9-5. GPIO Programming Model Flow (Part 2)

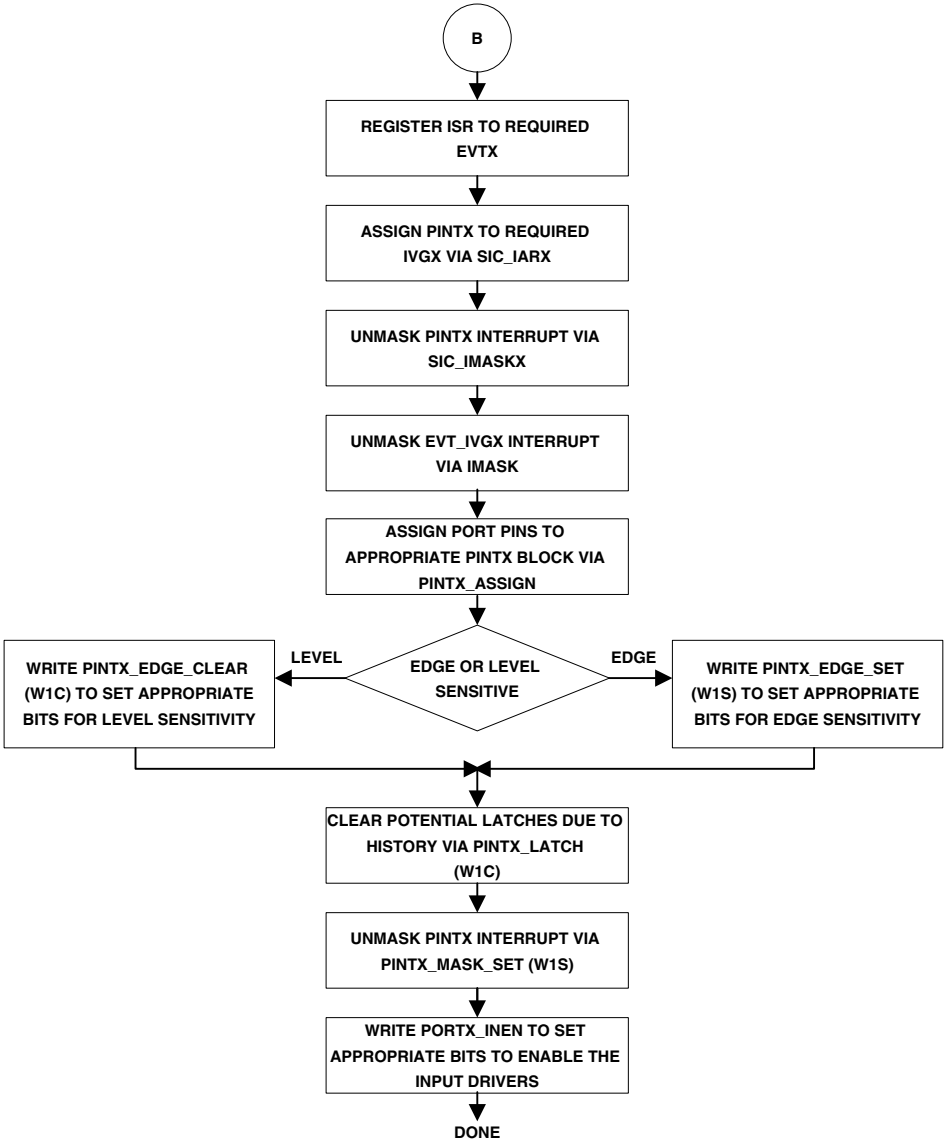


Figure 9-6. GPIO Programming Model Flow (Part 3)

Port Registers

The general-purpose ports are programmed using memory-mapped registers.

Table 9-13 and Table 9-14 on page 9-40 list the registers for port control and pin interrupt programming.

Table 9-13. Port Control Registers (Multiplexing and GPIO)

Address Offset	Register Name	Description	Notes
0xFFC0 14C0	PORTA_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 14C4	PORTA	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 14C8	PORTA_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 14CC	PORTA_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 14D0	PORTA_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 14D4	PORTA_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 14D8	PORTA_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 14DC	PORTA_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 14E0	PORTB_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 14E4	PORTB	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 14E8	PORTB_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 14EC	PORTB_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 14F0	PORTB_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 14F4	PORTB_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 14F8	PORTB_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 14FC	PORTB_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 1500	PORTC_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 1504	PORTC	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 1508	PORTC_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 150C	PORTC_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 1510	PORTC_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 1514	PORTC_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 1518	PORTC_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 151C	PORTC_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 1520	PORTD_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 1524	PORTD	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 1528	PORTD_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 152C	PORTD_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 1530	PORTD_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 1534	PORTD_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1538	PORTD_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 153C	PORTD_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 1540	PORTE_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 1544	PORTE	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 1548	PORTE_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 154C	PORTE_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 1550	PORTE_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 1554	PORTE_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 1558	PORTE_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 155C	PORTE_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 1560	PORTF_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1564	PORTF	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 1568	PORTF_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 156C	PORTF_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 1570	PORTF_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 1574	PORTF_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 1578	PORTF_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 157C	PORTF_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 1580	PORTG_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 1584	PORTG	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 1588	PORTG_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 158C	PORTG_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1590	PORTG_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 1594	PORTG_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 1598	PORTG_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 159C	PORTG_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 15A0	PORTH_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 15A4	PORTH	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 15A8	PORTH_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 15AC	PORTH_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 15B0	PORTH_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 15B4	PORTH_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 15B8	PORTH_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 15BC	PORTH_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 15C0	PORTI_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 15C4	PORTI	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000
0xFFC0 15C8	PORTI_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 15CC	PORTI_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 15D0	PORTI_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 15D4	PORTI_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 15D8	PORTI_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 15DC	PORTI_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000
0xFFC0 15E0	PORTJ_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45	R/W Reset = 0x0000
0xFFC0 15E4	PORTJ	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W Reset = 0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 15E8	PORTJ_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1S Reset = 0x0000
0xFFC0 15EC	PORTJ_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51	R/W1C Reset = 0x0000
0xFFC0 15F0	PORTJ_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1S Reset = 0x0000
0xFFC0 15F4	PORTJ_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48	R/W1C Reset = 0x0000
0xFFC0 15F8	PORTJ_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50	R/W Reset = 0x0000
0xFFC0 15FC	PORTJ_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46	R/W Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers

Address Offset	Register Name	Description	Notes
0xFFC0 1400	PINT0_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1S Reset = 0x0000 0000
0xFFC0 1404	PINT0_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1C Reset = 0x0000 0000
0xFFC0 1408	PINT0_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 140C	PINT0_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63	R/W Reset = 0x0000 0101
0xFFC0 1410	PINT0_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1S Reset = 0x0000 0000
0xFFC0 1414	PINT0_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1C Reset = 0x0000 0000
0xFFC0 1418	PINT0_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1S Reset = 0x0000 0000
0xFFC0 141C	PINT0_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1C Reset = 0x0000 0000
0xFFC0 1420	PINT0_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60	RO Reset = 0x0000 0000
0xFFC0 1424	PINT0_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000
0xFFC0 1430	PINT1_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1S Reset = 0x0000 0000
0xFFC0 1434	PINT1_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1C Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1438	PINT1_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000
0xFFC0 143C	PINT1_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63	R/W Reset = 0x0101 0000
0xFFC0 1440	PINT1_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1S Reset = 0x0000 0000
0xFFC0 1444	PINT1_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1C Reset = 0x0000 0000
0xFFC0 1448	PINT1_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1S Reset = 0x0101 0000
0xFFC0 144C	PINT1_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1C Reset = 0x0000 0000
0xFFC0 1450	PINT1_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60	RO Reset = 0x0000 0000
0xFFC0 1454	PINT1_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000
0xFFC0 1460	PINT2_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1S Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1464	PINT2_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1C Reset = 0x0000 0000
0xFFC0 1468	PINT2_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000
0xFFC0 146C	PINT2_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63	R/W Reset = 0x0000 0101
0xFFC0 1470	PINT2_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1S Reset = 0x0000 0000
0xFFC0 1474	PINT2_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1C Reset = 0x0000 0000
0xFFC0 1478	PINT2_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1S Reset = 0x0000 0000
0xFFC0 147C	PINT2_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1C Reset = 0x0000 0000
0xFFC0 1480	PINT2_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60	RO Reset = 0x0000 0000
0xFFC0 1484	PINT2_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 1490	PINT3_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1S Reset = 0x0000 0000
0xFFC0 1494	PINT3_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54	R/W1C Reset = 0x0000 0000
0xFFC0 1498	PINT3_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000
0xFFC0 149C	PINT3_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63	R/W Reset = 0x0202 0303
0xFFC0 14A0	PINT3_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1S Reset = 0x0000 0000
0xFFC0 14A4	PINT3_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58	R/W1C Reset = 0x0000 0000
0xFFC0 14A8	PINT3_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1S Reset = 0x0000 0000
0xFFC0 14AC	PINT3_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61	R/W1C Reset = 0x0000 0000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	Description	Notes
0xFFC0 14B0	PINT3_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60	RO Reset = 0x0000 0000
0xFFC0 14B4	PINT3_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55	R/W1C Reset = 0x0000 0000

Port Multiplexing Registers

The port multiplexing registers are described in the following sections:

- "Port x Function Enable (PORTx_FER) Registers" on page 9-45
- "Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46

For information on using these registers, see "Pin Multiplexing Scheme" on page 9-4.

Port x Function Enable (PORTx_FER) Registers

After reset, all pins default to GPIO mode (See Figure 9-7). Setting a bit in the port function enable registers enables a peripheral module to take ownership of the pin. The function enable bits impact output control only. Regardless of the setting of the function enable bits, both GPIO and peripherals can still sense the pin input. Once a function is enabled, it is up to the `PORTx_MUX` registers as to which peripheral takes control.

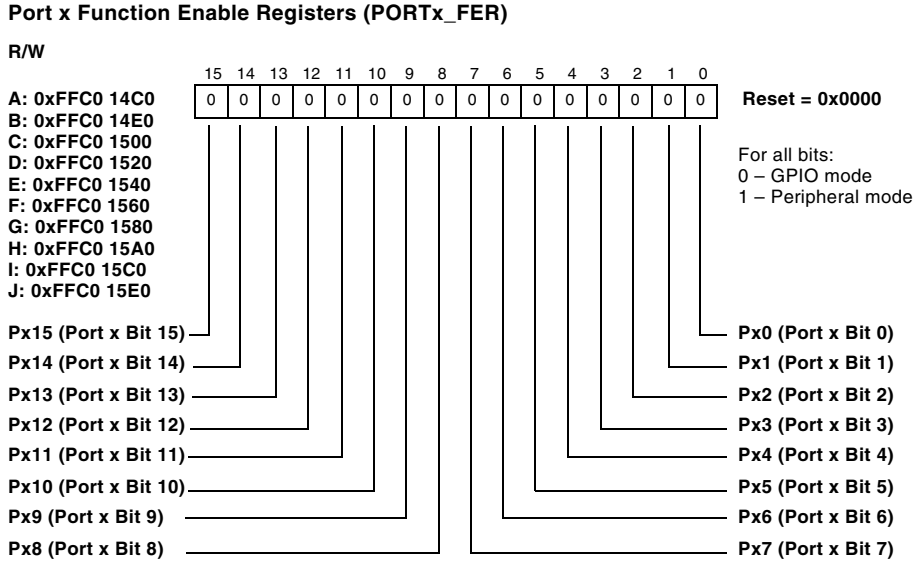


Figure 9-7. Port x Function Enable Registers

Port Multiplexer Control (PORTx_MUX) Registers

The multiplexer controls which peripheral takes ownership of a pin, if not in GPIO mode. Some ports have up to four different functions, while others have just a single function. Two bits are required to describe every multiplexer on an individual pin-by-pin scheme.

As a result, `PORTx_MUX` registers are 32 bits wide. Bit 0 and Bit 1 control the multiplexer of Pin 0. Bit 2 and Bit 3 control the multiplexer of Pin 1. Bit 30 and Bit 31 control the multiplexer of Pin 15.

The value of any `MUXy` bit has no affect on the port pins when the associated `Pxy` bit in the `PORTx_FER` registers is 0. Even if a port has only one function, the `PORTx_MUX` register is still present. For single function ports (no multiplexing is needed), leave the `MUXy` bits at 0 (default).

Normally, the PORTx_MUX register is accessed by 32-bit load/store instructions over the PAB bus (See Figure 9-8). The lower 16 bits can be accessed faster by 16-bit operations, alternately.

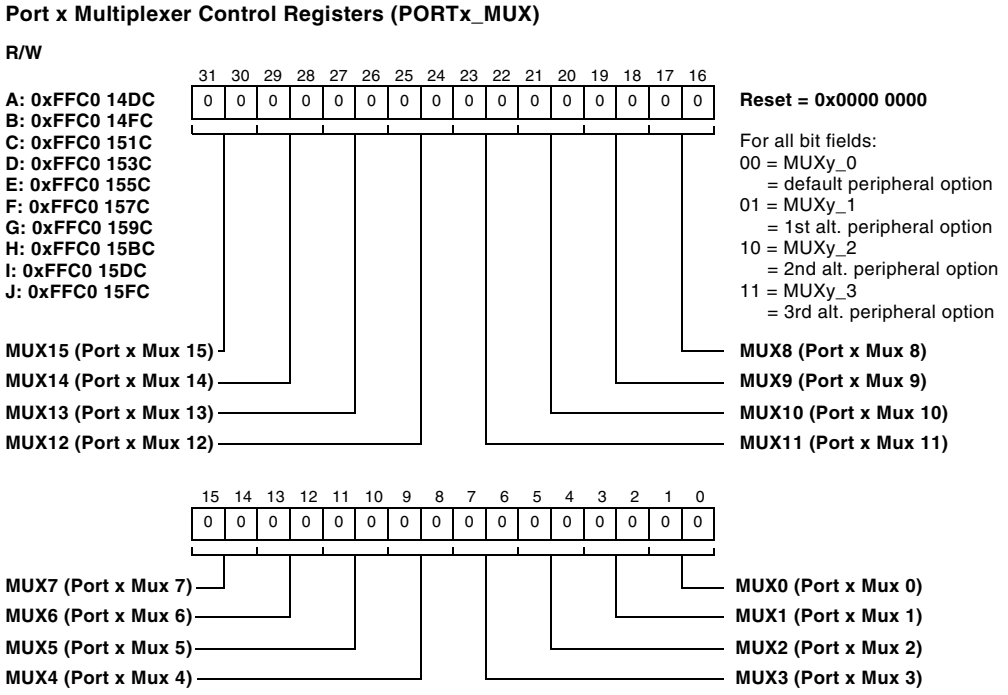


Figure 9-8. Port x Multiplexer Control Registers

GPIO Registers

The general-purpose I/O registers are described in the following sections.

- "Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
- "Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
- "Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51

For information on using these registers, see "GPIO Functionality" on page 9-24.

Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers

The direction registers control the output drivers of the GPIOs (See Figure 9-9 and Figure 9-10)). If set, the output driver is enabled and the GPIO is in output mode. If cleared as by default, the output driver is disabled. Note that the input driver is not enabled by default.

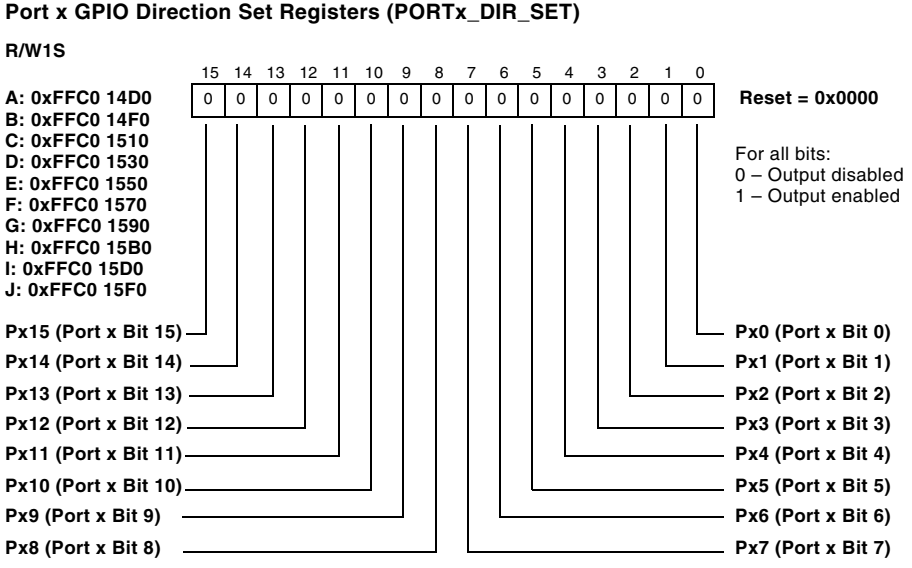


Figure 9-9. Port x GPIO Direction Set Registers

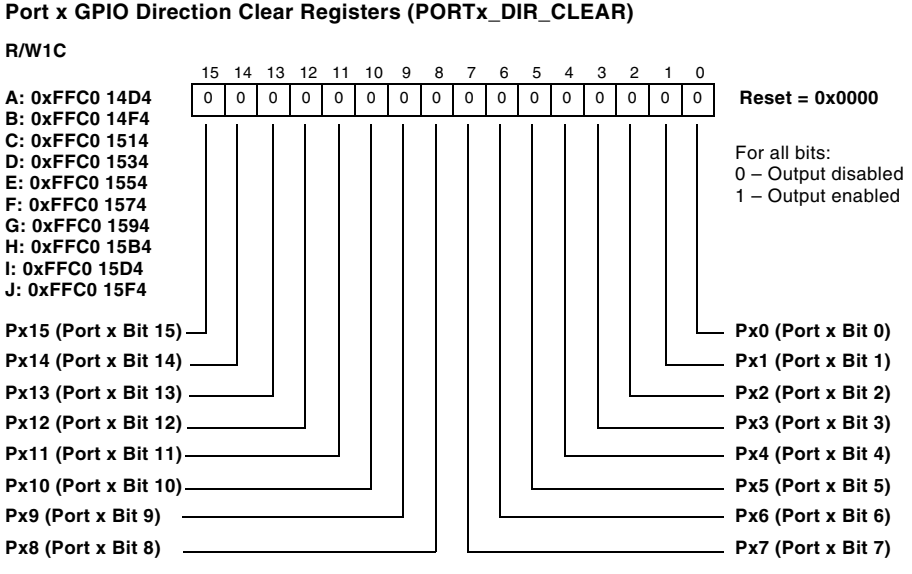


Figure 9-10. Port x GPIO Direction Clear Registers

Port x GPIO Input Enable (PORTx_INEN) Registers

By default, the input drivers are disabled after reset. To use a pin in GPIO input mode, the input driver must be enabled by writing a "1" to the PORTx_INEN register. If the input is enabled, reads from the PORTx/PORTx_SET/PORTx_CLEAR ports return the state of the pins.

However, the state of the output is not overwritten by the input (See Figure 9-11). It is altered by software writes only. Input and output drivers can be enabled at the same time. In this case, a read of the data register returns the true value of the data register and not the pin state.

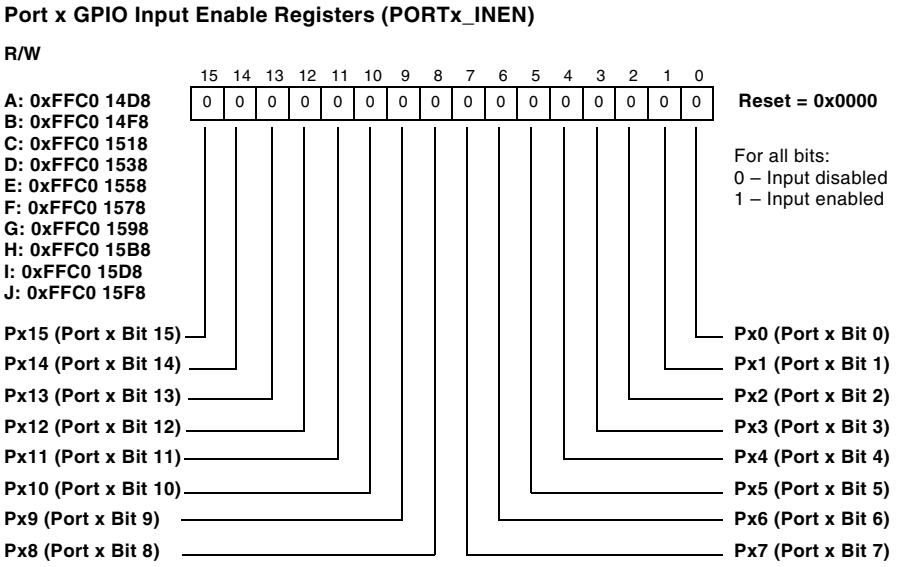


Figure 9-11. Port x GPIO Input Enable Registers

Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers

This group of registers controls the state of GPIO pins in output mode. Writes to the PORTx register impact the state of all pins of the port that are in output mode, for instance, that have their output driver enabled by the PORTx_DIR_SET and PORTx_DIR_CLEAR registers. The PORTx_SET and PORTx_CLEAR registers enable the software to set or clear specific pins without impacting other pins of the port.

When the input driver is enabled by the PORTx_INEN register, reads from any of the three registers return the state of the respective pins (See Table 9-12 on page 9-23 through Table 9-14 on page 9-40). When the input driver is not enabled as by default, reads from any of the registers return the value previously written to the registers.

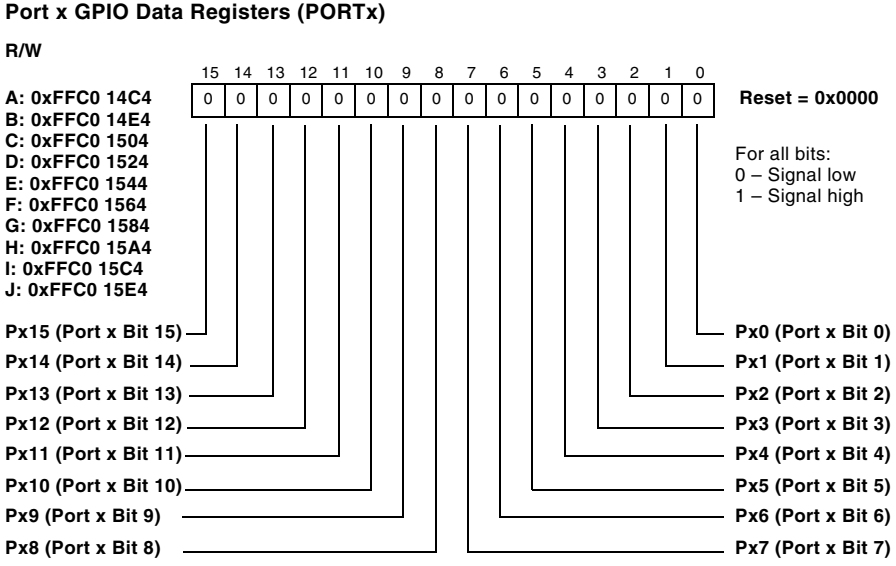


Figure 9-12. Port x GPIO Data Registers

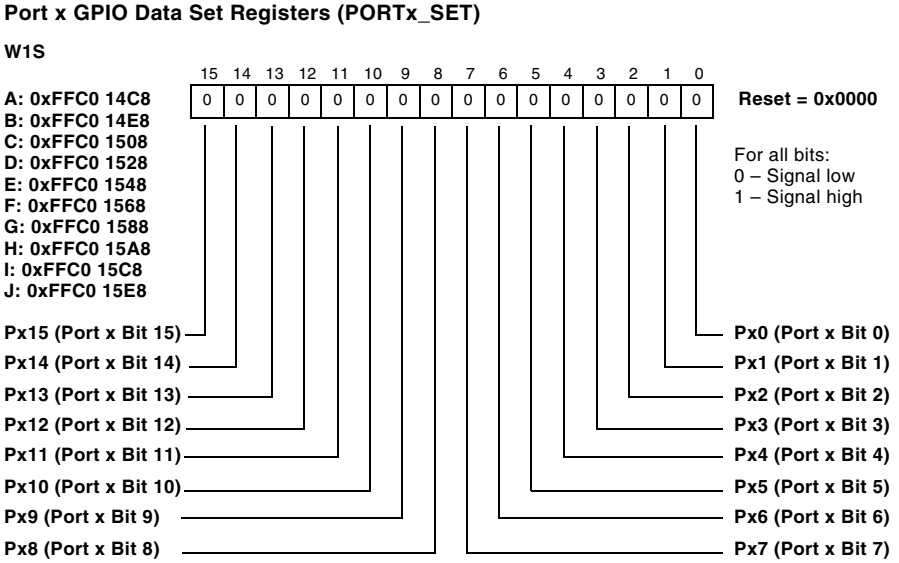


Figure 9-13. Port x GPIO Data Set Registers

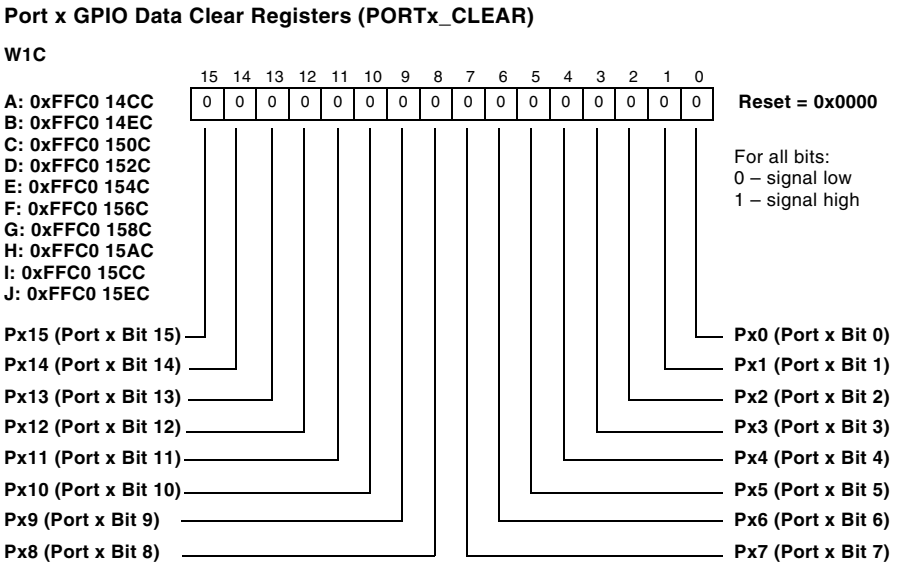


Figure 9-14. Port x GPIO Data Clear Registers

Pin Interrupt Registers

All PINTx registers are 32 bits wide and can be accessed by 32-bit load/store instructions. They also support 16-bit type of operation where the upper 16 bits are ignored and the application uses the lower 16 bits only. Consequently, all PINTx registers support 32-bit PAB accesses as well as 16-bit PAB accesses for the lower half words. Applications may use faster 16-bit accesses as long as they do not require functionality of upper register halves.

The pin interrupt registers are described in the following sections.

- "Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54
- "Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
- "Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
- "Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60
- "Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
- "Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63

Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs

The pairs of W1S and W1C registers enable interrupt functionality on respective pins (See Figure 9-15 and Figure 9-16). Setting a bit enables the interrupt. After reset, all bits are cleared. Note that the mask cannot be written directly by the PAB bus (no data register). Masks are controlled by W1S and W1C operations only.

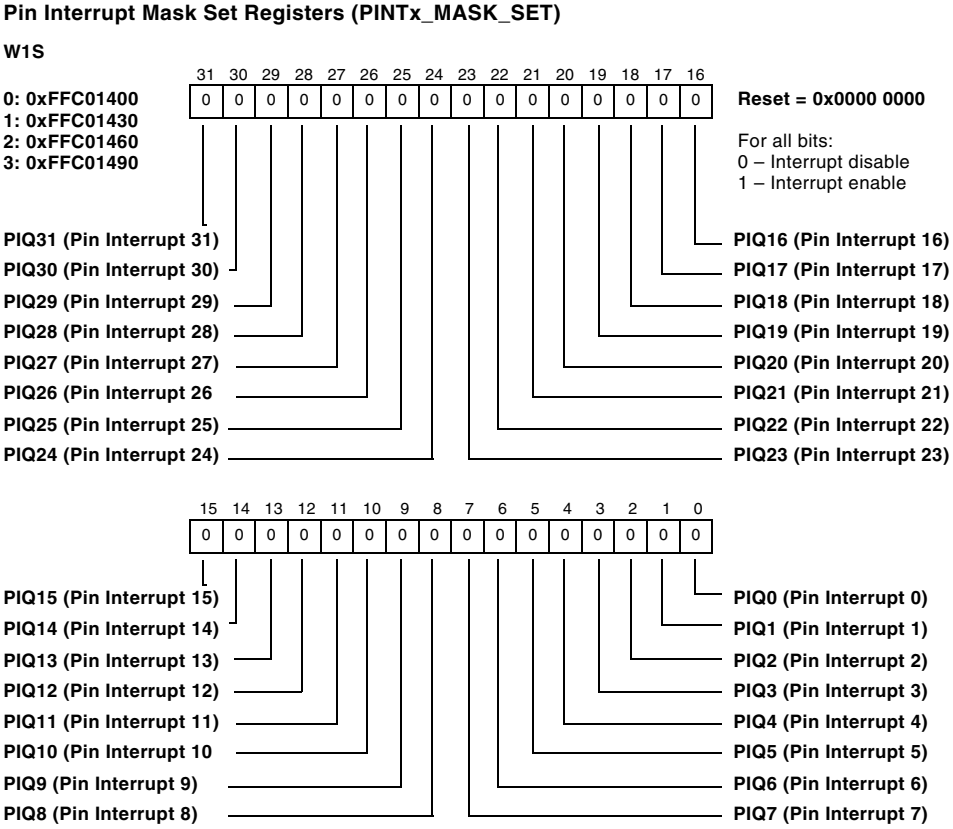


Figure 9-15. Pin Interrupt Mask Set Registers

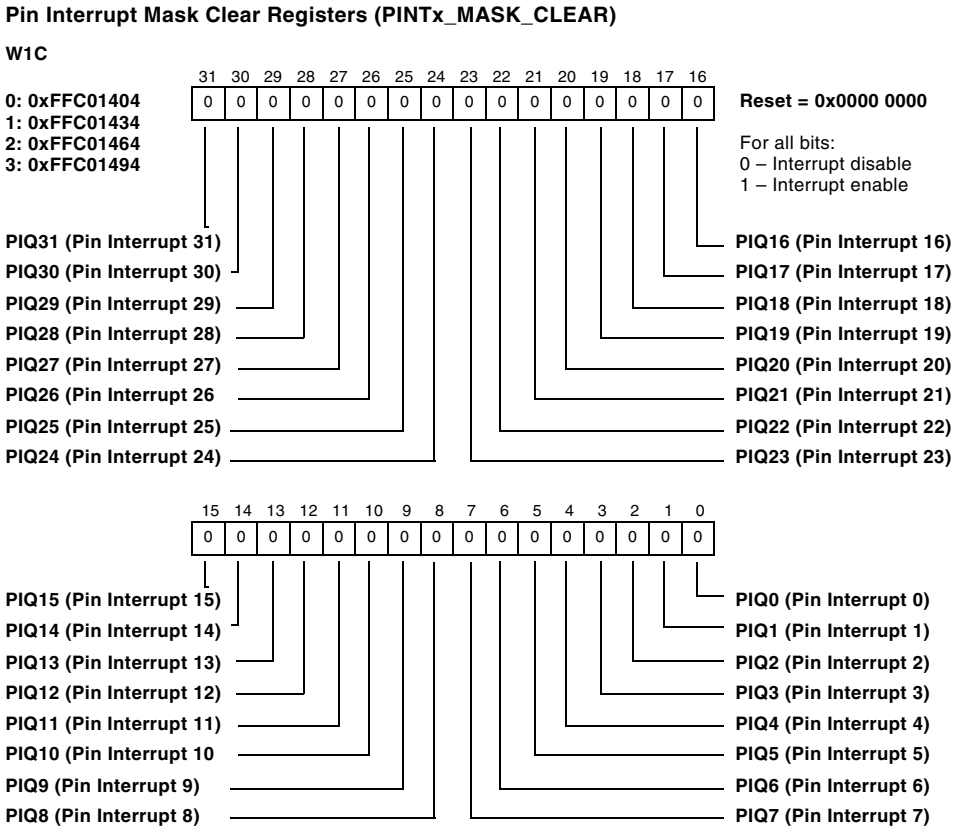


Figure 9-16. Pin Interrupt Mask Clear Registers

Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers

Both registers indicate whether an interrupt request is latched on the respective pin (See Figure 9-17). The PINTx_LATCH register is a latch that operates regardless of the interrupt masks. Bits of the PINTx_REQUEST register depend on the mask register. The PINTx_REQUEST register is a logical AND of the PINTx_LATCH register and the interrupt mask.

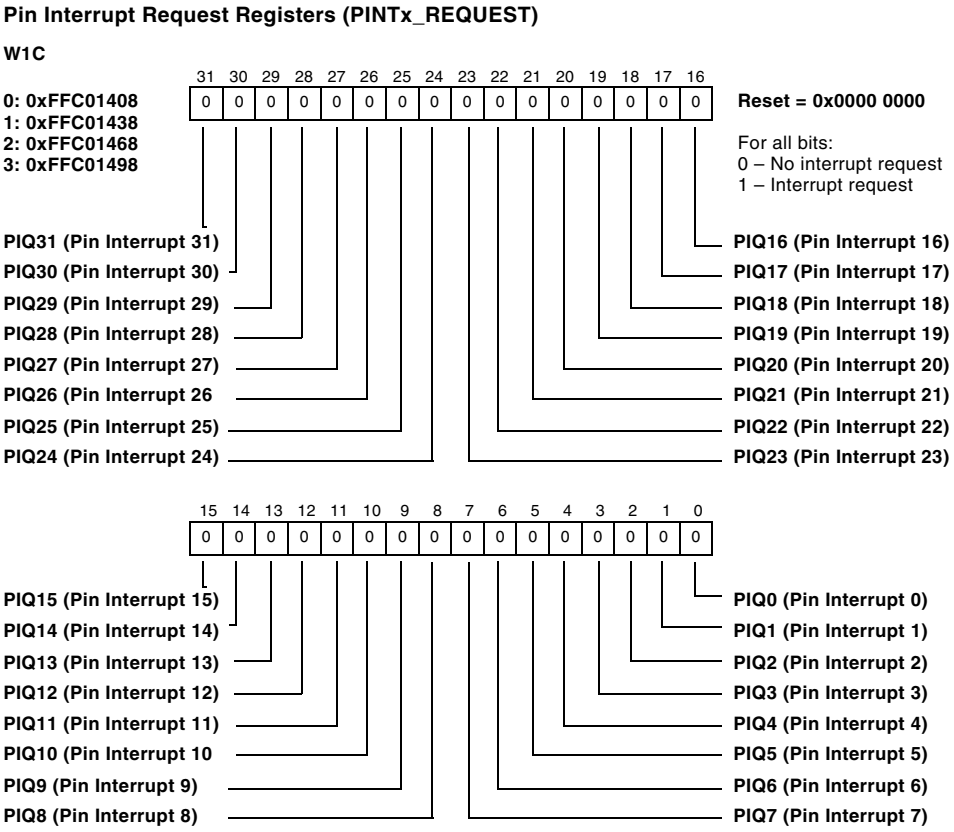


Figure 9-17. Pin Interrupt Request Registers

Having two separate registers here enables the user to interrogate certain pins in polling mode while others work in interrupt mode. The PINTx_LATCH registers can be used for edge detection or pin activity detection.

Both registers have W1C behavior (See Figure 9-18). Writing a 1 to either clears respective bits in both registers. For interrupt operation, the user may prefer to W1C the PINTx_REQUEST register (address still loaded in Px pointer). In polling mode it might be cleaner to W1C the PINTx_LATCH register.

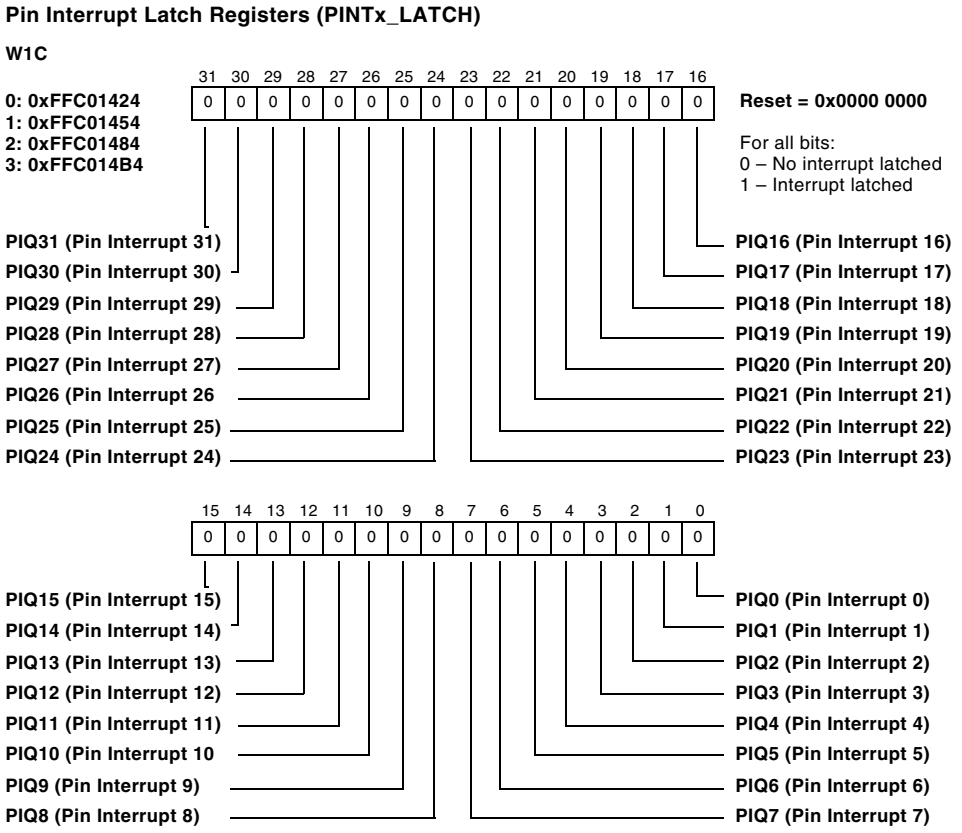


Figure 9-18. Pin Interrupt Latch Registers

Regardless whether in edge-sensitive mode or level-sensitive mode, `PINTx_LATCH` bits are never cleared by hardware except at system reset. Even in level-sensitive mode, the `PINTx_LATCH` register functions as latch.

Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs

This register pair controls whether the individual interrupts are edge-sensitive or level-sensitive (See Figure 9-19). Level sensitivity is default. After a W1S operation to the `PINTx_EDGE_SET` register, edge sensitivity for the interrupt is enabled.

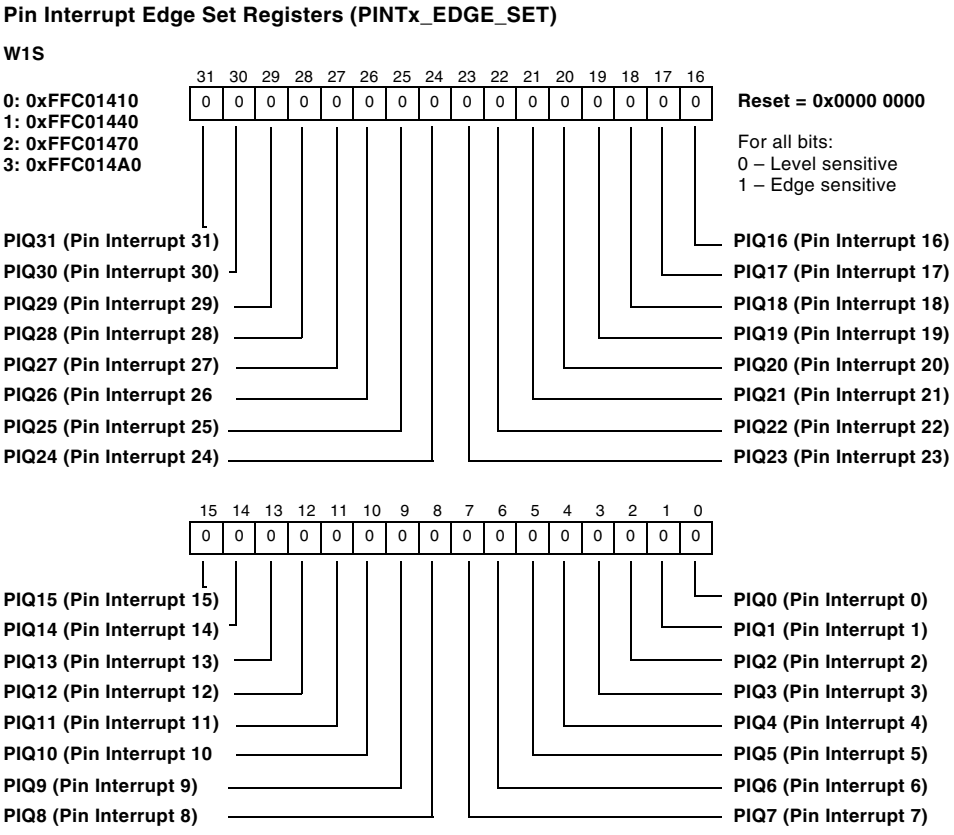


Figure 9-19. Pin Interrupt Edge Set Registers

After a W1C operation to the PINTx_EDGE_CLEAR register, edge sensitivity for the interrupt is disabled, and the interrupt returns to level sensitivity.

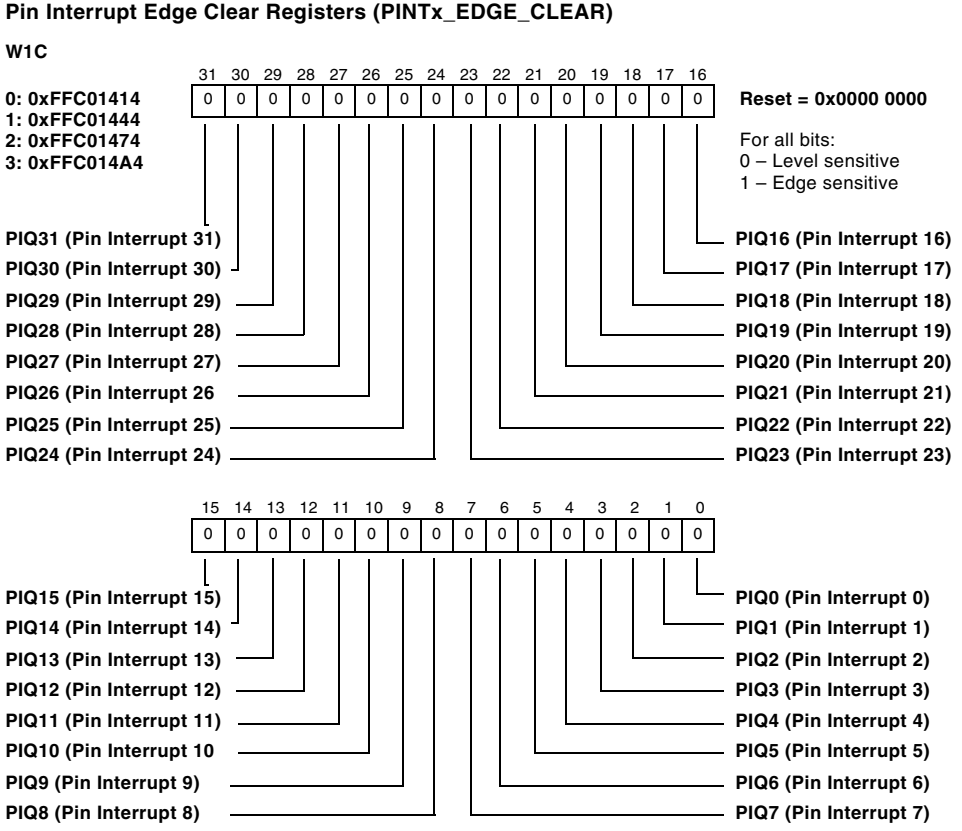


Figure 9-20. Pin Interrupt Edge Clear Registers

An interrupt request is generated on either edge of a signal, if PINTx_ASSIGN settings forward a signal to two PINTx channel bits, and one channel inverts the signal polarity.

Pin Interrupt Pin State (PINTx_PINSTATE) Register

The pin interrupt pin state registers enable the service routine to read the current state of the pin without reading from GPIO space (See Figure 9-21). If there was an edge-sensitive interrupt, the service routine can check whether the state of the pin is still high or turned low.

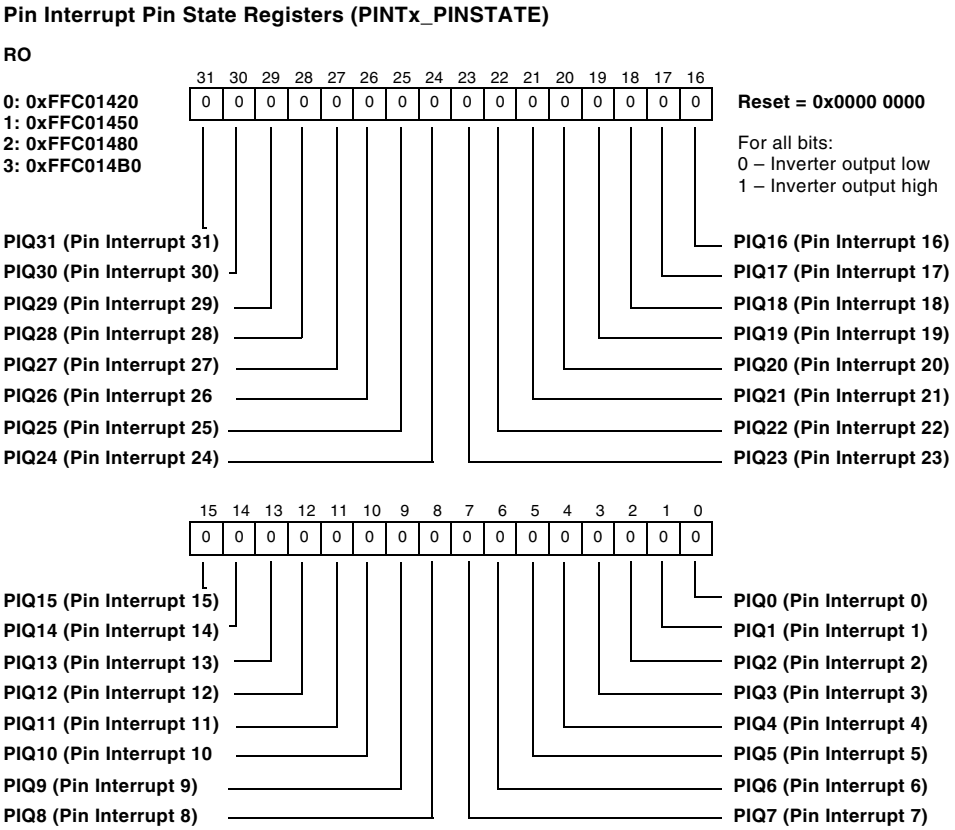


Figure 9-21. Pin Interrupt Pin State Registers

Note that the content of the `PINTx_PINSTATE` register depends on the polarity setting of the `PINTx_INVERT_SET`/`PINTx_INVERT_CLEAR` registers.

Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers

These register pairs control the inverters at the input of the module (See Figure 9-22). After reset, the inverters are cleared and the PINTx_PINSTATE bits contain an exact copy of the pin state. With the inverters on, PINTx_PINSTATE register reads the inverted/negated pin state.

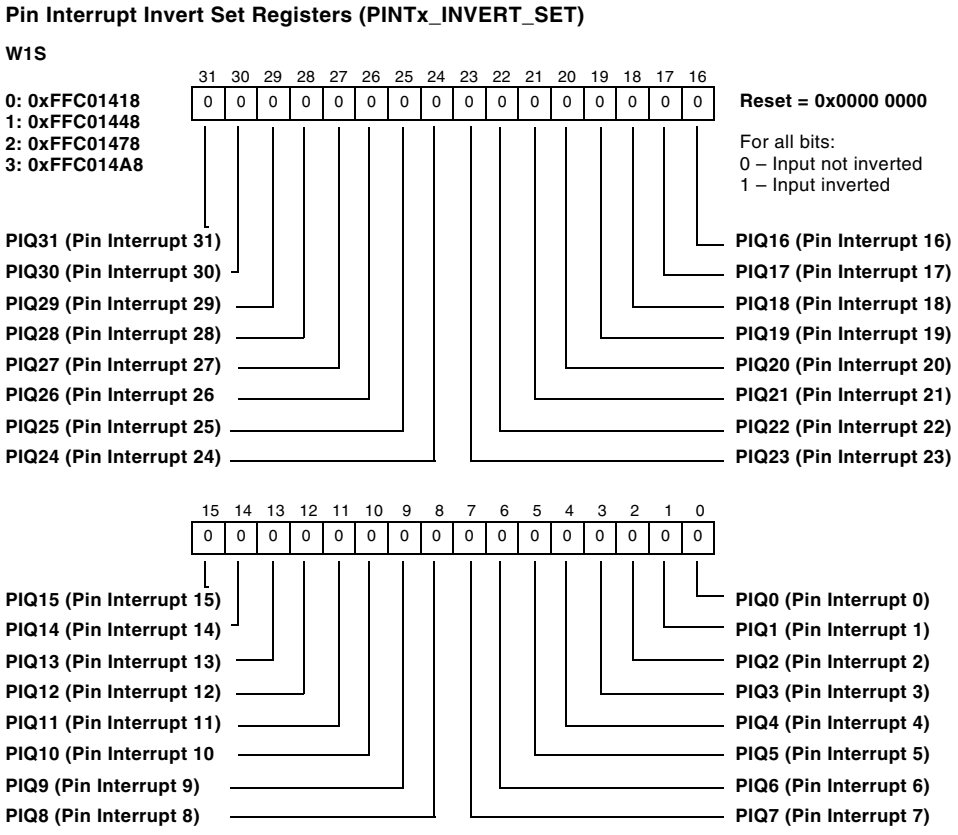


Figure 9-22. Pin Interrupt Invert Set Registers

In level-sensitive mode, the interrupt is active when PINTx_PINSTATE is logical “1”. For instance, when the pin is high and the inverter is off, or when the pin is low and the inverter is on.

In edge-sensitive mode, the rising edges are latched when the inverter is off (See Figure 9-23). With the inverter on, falling edges generate the interrupt.

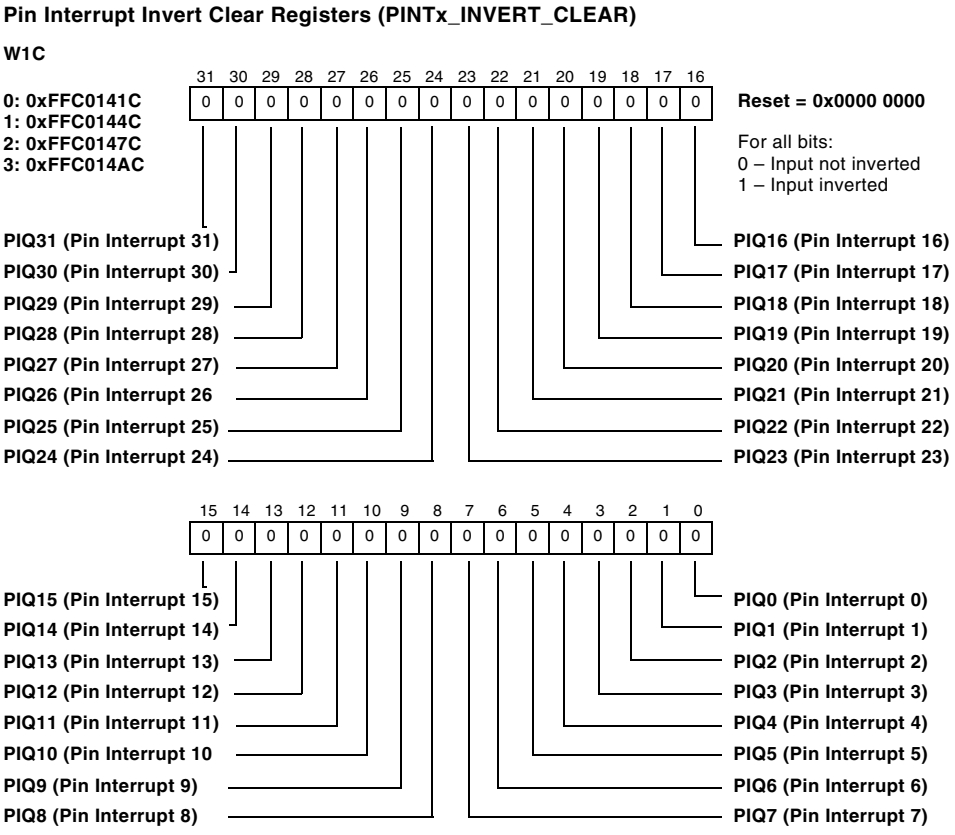


Figure 9-23. Pin Interrupt Invert Clear Registers

Pin Interrupt Assignment (PINTx_ASSIGN) Registers

The 32-bit pin interrupt assignment registers control the pin-to-interrupt assignment in a byte-wide manner. Unlike the other pin interrupt registers, the pin interrupt assignment registers do not consist of 32 individual bits. They consist of four control bytes each that function as a multiplexer control.

On ADSP-BF54x processor Blackfin processors, only three bits of each byte are populated. The other bits are reserved. Both PINT0 and PINT1 blocks can sense to signals of port A and port B. The lower eight pins of port A or port B for example, can be forwarded to either the byte 0 or byte 2 of the pin interrupt registers. Similarly, the upper eight pins can be forwarded to byte 1 or byte 3 of the pin interrupt registers. Both PINT2 and PINT3 blocks can sense to signals of port C to port J. The lower eight pins of any of those ports can be mapped to byte 0 or byte 2 of the pin interrupt registers. Similarly, the upper eight pins of any two ports can be mapped to byte 1 and byte 3.

Figure 9-24 shows the PINT0_ASSIGN register.

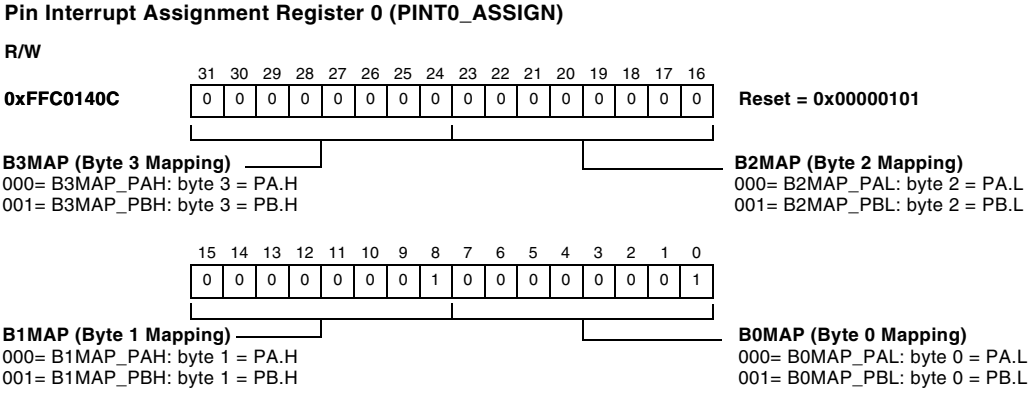


Figure 9-24. Pin Interrupt Assignment Register 0

Figure 9-25 shows the PINT1_ASSIGN register.

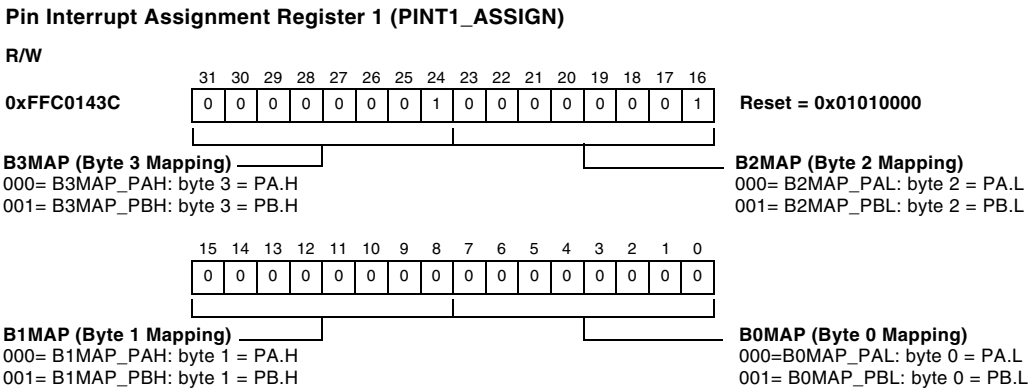


Figure 9-25. Pin Interrupt Assignment Register 1

Figure 9-26 shows the PINT2_ASSIGN register.

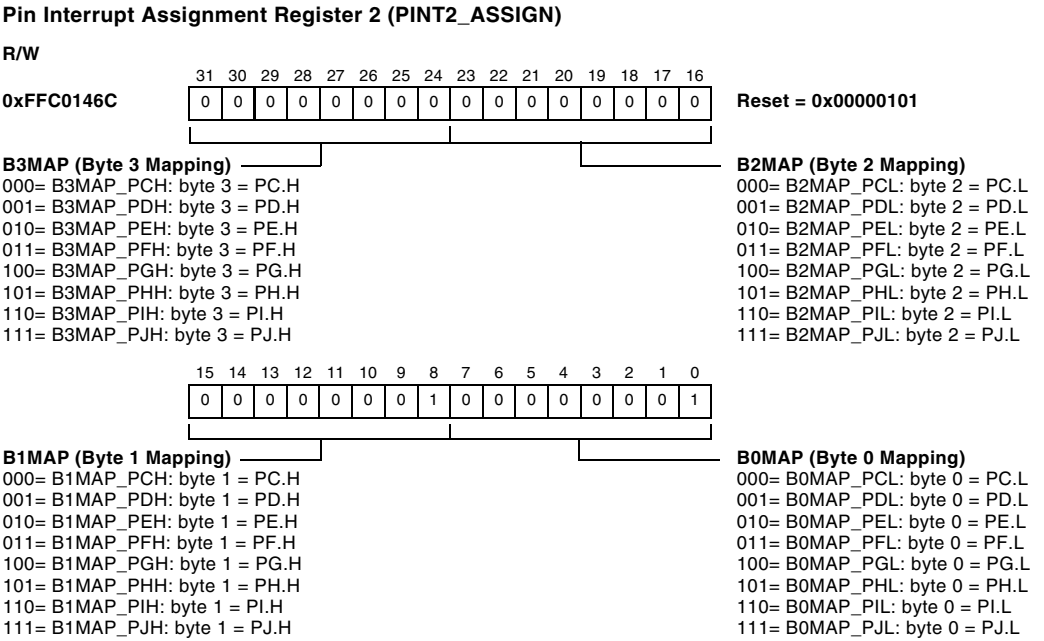


Figure 9-26. Pin Interrupt Assignment Register 2

Figure 9-27 shows the PINT3_ASSIGN register.

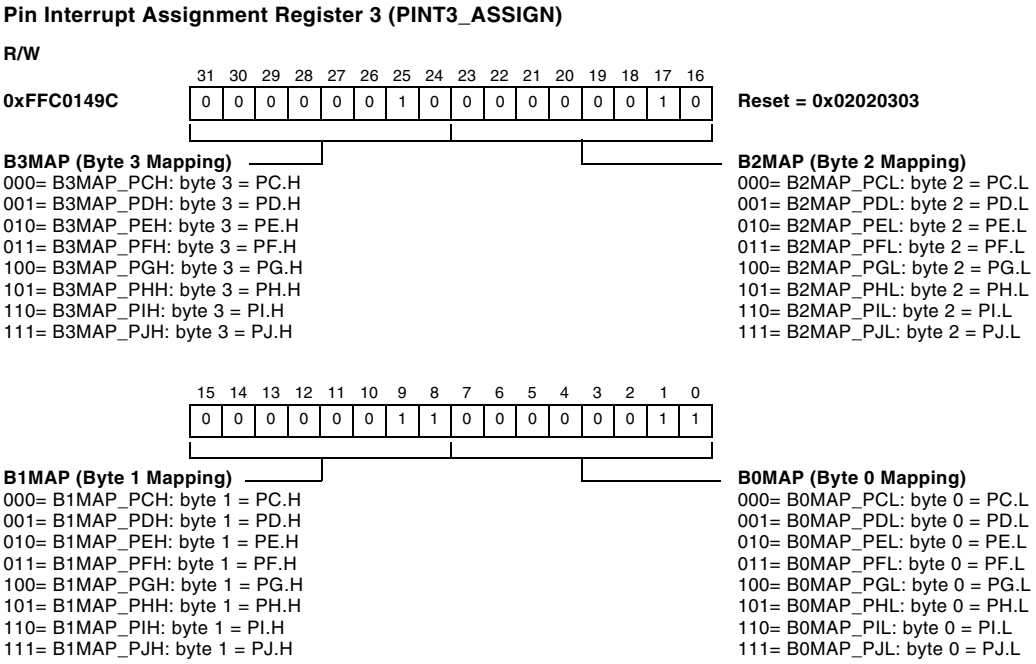


Figure 9-27. Pin Interrupt Assignment Register 3

Programming Examples

Listing 9-1 illustrates how to enable the output drivers of the port pins PG6 and PG7 on port G. The pins are toggled afterward.

Listing 9-1. Output Driver Enable

```
/* enable GPIO mode. */
/* This is optional as all PORTx_FER register
   are cleared by default after reset */
P5.H = hi(PORTG_FER);
```

```
P5.L = lo(PORTG_FER);
R7 = PG7 | PG6 (z);
R6 = ~R7;
R5 = w[P5](z);
R5 = R5 & R6;
w[P5] = R5;

/* start with PG7=0 and PG6=1 */
P5.L = lo(PORTG);
R5 = w[P5] (z);
R5 = R5 & R6;
bitset(R5, bitpos(PG6));
w[P5] = R5;

/* enable output drivers */
P5.L = lo(PORTG_DIR_SET);
w[P5] = R7;

...

/* clear PG6 */
P5.L = lo(PORTG_CLEAR);
R5 = PG6;
w[P5] = R5;

/* set PG7 */
P5.L = lo(PORTG_CLEAR);
R5 = PG7;
w[P5] = R5;
```

Note that the level of the GPIO flags can be defined before the output is enabled. With the separate set and clear ports of the data and direction registers multiple software threads can control their own pins individually.

Listing 9-2 programs the port pin PG8 on port G in open-drain mode. It assumes an external pull-up resistor. Once the PG8 bit is also set in the PORTG_INEN register reads from PORTG register return the actual state of the pin.

Listing 9-2. Open-Drain Mode Programming

```
/* set the internal flag to zero */
P5.H = hi(PORTG_CLEAR);
P5.L = lo(PORTG_CLEAR);
R5 = PG8 (z);
w[P5] = R5;

/* enable input driver */
P5.L = lo(PORTG_INEN);
P5.H = hi(PORTG_INEN);
R6 = w[P5] (z);
R6 = R5 | R6;
w[P5] = R6;

/* drive the PG8 pin low */
P5.L = lo(PORTG_DIR_SET);
w[P5] = R5;

...

/* three-state the PG8 pin again */
P5.L = lo(PORTG_DIR_CLEAR);
w[P5] = R5;
```

Listing 9-3 illustrates the pin interrupt functionality. The input pin PB8 in configured to request an IVG6 interrupt through the pin interrupt block PINT0 every time a raising edge is detected.

Listing 9-3. Pin Interrupt Functionality

```
#include <blackfin.h>

.section program;
.global _main;
_main:
    /* register interrupt service routines */
    R7.L = lo(_isr_PB8);
    R7.H = hi(_isr_PB8);
    P5.L = lo(EVT7);
    P5.H = hi(EVT7);
    [P5] = R7;

    /* interrupt assignment PINT0 => IVG7 */
    R7.L = lo(0xFFFF0FFF);
    R7.H = hi(0xFFFF0FFF);
    P5.L = lo(SIC_IAR2);
    P5.H = hi(SIC_IAR2);
    [P5] = R7;

    /* interrupt unmasking */
    R7.L = lo(IRQ_PINT0);
    R7.H = hi(IRQ_PINT0);
    P5.L = lo(SIC_IMASK0);
    P5.H = hi(SIC_IMASK0);
    [P5] = R7;
    R7 = EVT_IVG7;
    P5.L = lo(IMASK);
    P5.H = hi(IMASK);
    [P5] = R7;

    /* enable input drivers for push-button on Port B */
```

```
    /* pin can be also output or input enabled by other functions
*/
    P5.L = lo(PORTB_INEN);
    P5.H = hi(PORTB_INEN);
    R6 = w[P5] (z);
    R7 = PB8 (z);
    R6 = R6 | R7;
    w[P5] = R6;

    /* assign PB8 to PINT0 byte 1 */
    P5.L = lo(PINT0_ASSIGN);
    P5.H = hi(PINT0_ASSIGN);
    R7.L = lo(B1MAP_PBH);
    R7.H = hi(B1MAP_PBH);
    [P5] = R7;

    /* set to raising edge sensitivity */
    R7.L = lo(PB8);
    R7.H = hi(PB8);
    P5.L = lo(PINT0_INVERT_CLEAR);
    P5.H = hi(PINT0_INVERT_CLEAR);
    [P5] = R7;
    P5.L = lo(PINT0_EDGE_SET);
    P5.H = hi(PINT0_EDGE_SET);
    [P5] = R7;

    /* W1C potential latches due to history */
    P5.L = lo(PINT0_LATCH);
    P5.H = hi(PINT0_LATCH);
    [P5] = R7;

    /* unmask interrupts */
    P5.L = lo(PINT0_MASK_SET);
    P5.H = hi(PINT0_MASK_SET);
```

```
    [P5] = R7;

    JUMP 0;
_main.end:
```

Listing 9-4 shows the fragments of an interrupt service routine that matches for Listing 9-3. The interrupt request can be cleared by W1C operation to either the PINT0_REQUEST or the PINT0_LATCH register.

Listing 9-4. Interrupt Service Routine Programming

```
_isr_PB8:
    [--SP] = ASTAT;
    [--SP] = (R7:5, P5:4);

    /* clear interrupt request early in the ISR*/
    P5.L = lo(PINT0_REQUEST);
    P5.H = hi(PINT0_REQUEST);
    R7 = PB8 (z);
    [P5] = R7;

    /* more service code goes to here */

    SSYNC;
    (R7:5, P5:4) = [SP++];
    ASTAT = [SP++];
    RTI;
_isr_PB8.end:
```

Listing 9-5 provides a C version of Listing 9-3 and Listing 9-4. Additionally, every interrupt event toggles the output on the PF6 GPIO pin.

Listing 9-5. Pin Interrupts and Interrupt Service in C

```
#include <blackfin.h>
#include <ccblkfn.h>
#include <sys/exception.h>

short dPattern;

/* interrupt service routine */
EX_INTERRUPT_HANDLER(IsrPB8)
{
    /* clear interrupt request */
    *pPINT0_REQUEST = PB8;

    /* toggle output on PG6 */
    if (dPattern & PG6)
    {
        *pPORTG_CLEAR = PG6;
    }
    else
    {
        *pPORTG_SET = PG6;
    }
    dPattern^= PG6;
}

void main (void)
{
    /* register interrupt routine */
    register_handler(ik_ivg7, IsrPB8);
    /* assign PINT0 interrupt to IVG7 */
    *pSIC_IAR2 = 0xFFFF0FFFL;
    *pSIC_IMASK0 = IRQ_PINT0;
```

```
    /* enable the PB8 input driver */
    *pPORTB_INEN = PB8;

    /* assign PB8 to PINT0 byte 1 */
    *pPINT0_ASSIGN = B1MAP_PBH;

    /* set to raising edge sensitivity */
    *pPINT0_INVERT_CLEAR = PB8;
    *pPINT0_EDGE_SET = PB8;

    /* W1C potential latches due to history */
    *pPINT0_LATCH = PB8;

    /* unmask interrupts */
    *pPINT0_MASK_SET = PB8;

    /* initialize PG6 to high */
    *pPORTG_SET = PG6;
    *pPORTG_DIR_SET = PG6;
    dPattern = PG6;

    while (1);
}
```

Listing 9-6 illustrates how to control the port multiplexing. In the example, Port H is configured to provide the following signals: `UART1 TX` and `RX`, `TMR8`, `CDG` and `CUD`, `DMAR0` and `DMAR1` and `A4` to `A9`. `PH7` operates in GPIO mode.

Listing 9-6. Port Multiplexing Example

```
P5.H = hi(PORTH_FER);
P5.l = lo(PORTH_FER);
R5.L = PH15 | PH14 | PH13 | PH12 | PH11 | PH10 | PH9 |PH8
       | nPH7 | PH6 | PH5 | PH4 | PH3 | PH2 | PH1 | PH0;
w[P5] = R5;

P5.H = hi(PORTH_MUX);
P5.L = lo(PORTH_MUX);
R5.H = MUX15_0 |MUX14_0 |MUX13_0 | MUX12_0
       | MUX11_0 | MUX10_0 | MUX9_0 | MUX8_0;
R5.L = MUX7_0 |MUX6_1 | MUX5_1 | MUX4_2
       | MUX3_2 | MUX2_1 | MUX1_0 | MUX0_0;
[P5] = R5;

/*For the second part where the PORTH_MUX register is configured,
a more compact syntax can be used as shown below.*/
P5.H=hi(PORTH_MUX);
P5.L=lo(PORTH_MUX);
R5.H=hi(MUX(0,0, 0,0,0,0,0,0, 0, 1,1, 2,2, 1, 0,0));
R5.L=lo(MUX(0,0, 0,0,0,0,0,0, 0, 1,1, 2,2, 1, 0,0));
 [P5] = R5;
```

10 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose timer modules and includes the following sections:

- "Overview and Features" on page 10-1
- "Interface Overview" on page 10-3
- "Description of Operation" on page 10-6
- "Modes of Operation" on page 10-13
- "Programming Model" on page 10-35
- "Timer Registers" on page 10-37
- "Programming Examples" on page 10-53

Overview and Features

The ADSP-BF544, ADSP-BF547, ADSP-BF548, and ADSP-BF549 Blackfin processors feature two general-purpose timer modules that contain eleven identical 32-bit timers. The ADSP-BF542 processors feature only one timer module with eight timers. Every timer can operate in various operating modes on individual configuration. Although the timers operate completely independent from each other, all of them can be started and stopped simultaneously for synchronous operation.

Features

The general-purpose timers support the following operating modes:

- Singleshot mode for interval timing and single pulse generation
- Pulse-width modulation (PWM) generation with consistent update of period and pulse width values
- External signal capture mode with consistent update of period and pulse width values
- External event counter mode

Feature highlights include:

- Synchronous operation of all timers
- Consistent management of period and pulse width values
- Autobaud detection for CAN and both UART modules
- Period measurement for the GP counter module
- Graceful bit pattern termination when stopping
- Support for center-aligned PWM patterns
- Error detection on implausible pattern values
- All read and write accesses to 32-bit registers are atomic
- Every timer has its dedicated interrupt request output
- Unused timers can function as edge-sensitive pin interrupts

Interface Overview

Figure 10-1 shows the derivative-specific block diagram of the general-purpose timer module.

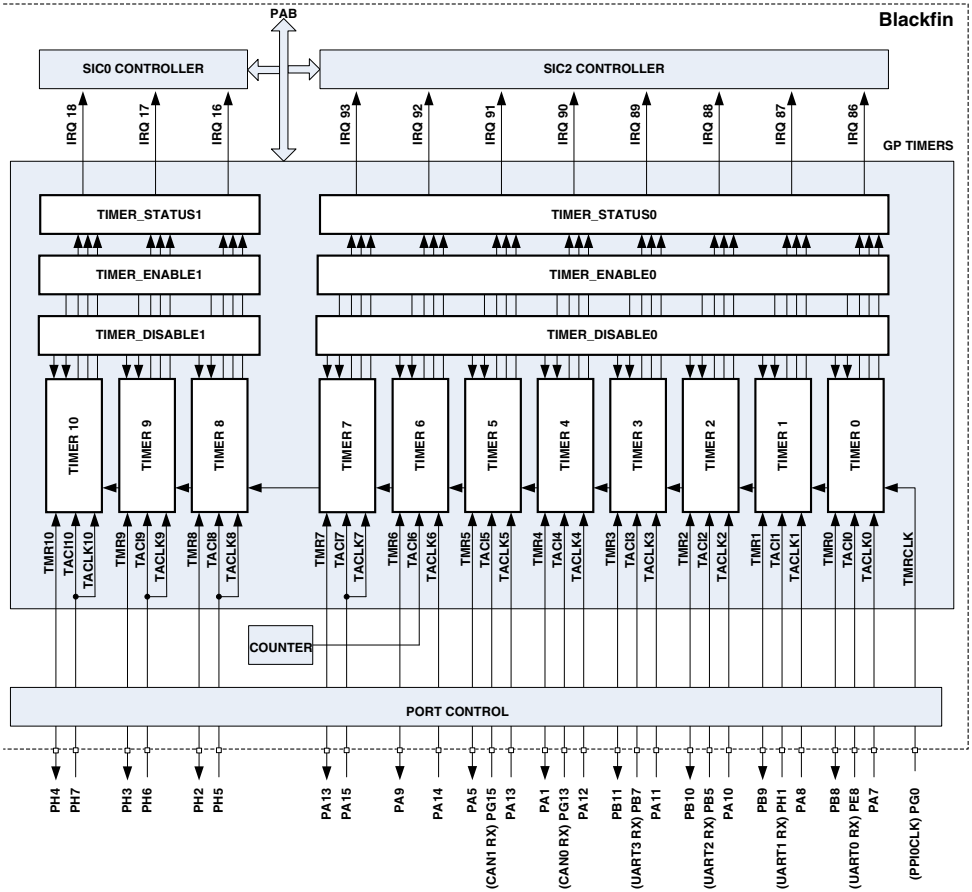


Figure 10-1. Timer Block Diagram

The timer module features a global infrastructure to control synchronous operation of all timers if required. The internal structure of the individual timers is illustrated by Figure 10-2, which shows the details of timer 0 representatively. The other timers have identical structure.

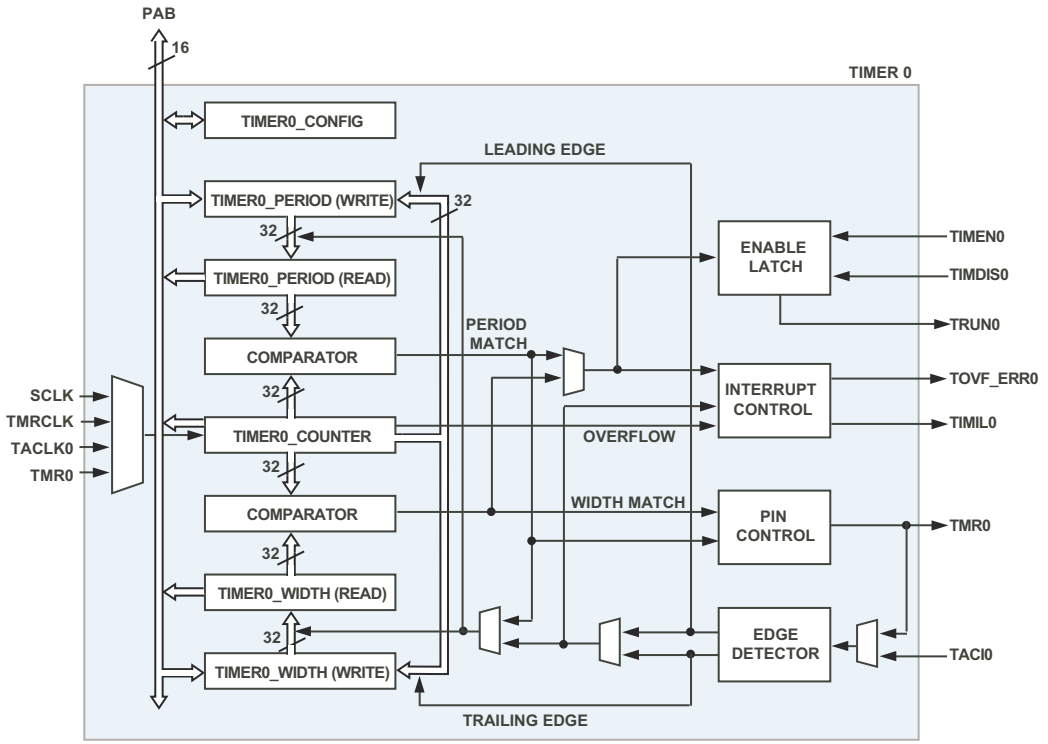


Figure 10-2. Internal Timer Structure

External Interface

Every timer has a dedicated TMRx pin that can be found on ports A, B, and H. 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.

Alternate clock (TACLKx) and capture (TACIx) inputs are found on ports A, B, E, G, and H. The TACLKx pins can alternatively clock the timers in PWM_OUT mode.

In WDTH_CAP mode, timers 0-5 feature TACIx inputs that can be used for bit rate detection on CAN and UART inputs. The TACI0-TACI3 pins connect, respectively, to the UART0-UART3 RX inputs. Additionally, the TACI4 input connects to the CAN0 RX input, and the TACI5 input connects to the CAN1 RX input. The TACI6 input senses to an output of the general purpose counter module and supports capturing of the event timing this way. The TACI7, TACI8, TACI9 and TACI10 inputs are available on pins for various purposes. TACIx inputs can be used with or without the respective UART or CAN peripheral enabled. If the peripheral is not enabled, the input drivers of the TACIx inputs must be explicitly enabled.

The TMRCLK input is another clock input common to all 11 timers. The EPPI0 unit is clocked by the same pin; therefore any of the timers can be clocked by EPPI0_CLK.

In order to enable TMRCLK, the PORTG_FER bit 0 must be set and input enable for GPIO bit 0 needs to be set in the PORTG_INEN register.

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is ($(2^{32}-1)$ / 133 MHz) = 32.2 seconds.

Clock and capture input pins are sampled every SCLK cycle. The duration of every low or high state must be one SCLK minimum. The maximum allowed frequency of timer input signals is SCLK/2.

Internal Interface

Timer registers are always accessed by the core through the 16-bit PAB bus. Hardware ensures that all read and write operations from and to 32-bit timer registers are atomic.

Every timer has its dedicated interrupt request output that connects to the SIC controller, for a total of 11 interrupt outputs.

Description of Operation

The core of every timer is a 32-bit counter, that can be interrogated through the read-only TIMERx_COUNTER register. Depending on operation mode, the counter is reset to either 0x0000 0000 or 0x0000 0001 when the timer is enabled. The counter always counts upward. Usually, it is clocked by SCLK. In PWM mode it can be clocked by the alternate clock input TACLKx, or the common timer clock input TMRCLK alternatively. In counter mode, the counter is clocked by edges on the TMRx input. The significant edge is programmable.

After 2^{32}-1 clocks, the counter overflows. In this case, this is reported by the overflow/error bit TOVF_ERRx in the TIMER_STATUSx registers. In PWM and counter mode, the counter is reset by hardware when its content reaches the values stored in the TIMERx_PERIOD register. In capture mode the counter is reset by leading edges on the input pin TMRx or TACIx. If enabled, these events cause the interrupt latch TIMILx in the TIMER_STATUSx registers to be set and issue a system interrupt request. The TOVF_ERRx and TIMILx latches are sticky and should be cleared by software using W1C operations to clear the interrupt request. Each global TIMER_STATUSx register is 32 bits wide. A single atomic 32-bit read can consistently report the status of all timers within a given TIMER_STATUSx register.

Before a timer can be enabled, its mode of operation is programmed in its timer-specific TIMERx_CONFIG register. Then, one or more timers are started by writing a 1 to the representative bits in one or more of the TIMER_ENABLEx registers.

The TIMER_ENABLEx registers can be used to enable some or all timers within a block simultaneously, through “write-1-to-set” control bits, one for each timer. Likewise, the TIMER_DISABLEx registers can be used to disable some or all timers within a block at the same time, through “write-1-to-clear” control bits. The TIMER_ENABLE0 and TIMER_DISABLE0 registers control timers 0-7, while the TIMER_ENABLE1 and TIMER_DISABLE1 registers control timers 8-10. Either the TIMER_ENABLEx or TIMER_DISABLEx register for a given timer block can be read back to check the enable status of the timers. A 1 indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMENx bit is set.

While the PWM mode is used to generate PWM patterns, the capture mode (WDTH_CAP) is designed to “receive” PWM signals. A PWM pattern is represented by a pulse width and a signal period. This is described by the TIMERx_WIDTH and TIMERx_PERIOD register pair. In capture mode these registers are read-only. Hardware always captures both values. Regardless of whether in PWM or capture mode, shadow buffers always ensure consistency between the TIMERx_WIDTH and TIMERx_PERIOD values. In PWM mode, hardware performs a plausibility check by the time the timer is enabled. In this case the error type is reported by the TIMERx_CONFIG register and signalled by the TOVF_ERRx bit.

Interrupt Processing

Each of the 11 timers can generate a single interrupt. The 11 resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The TIMER_STATUSx registers latch the timer interrupts to provide a means for software to determine the interrupt source.

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the IMASK and SIC_IMASKx registers. To poll the TIMILx bit without interrupt generation, set IRQ_ENA but leave the interrupt masked at the system level. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions as reported by the TOVF_ERRx bits.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same CEC interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, multiple timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMILx latch bits at once (for timers 0-7) by writing 0x000F 000F to the TIMER_STATUS0 register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMILx bit in the TIMER_STATUSx register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMILx clear command from the RTI instruction, an extra SSYNC instruction may need to be inserted. In EXT_CLK mode, reset the TIMILx bit in the TIMER_STATUSx register at the very beginning of the interrupt service routine to avoid missing any timer events. Figure 10-3 shows the timers interrupt structure.

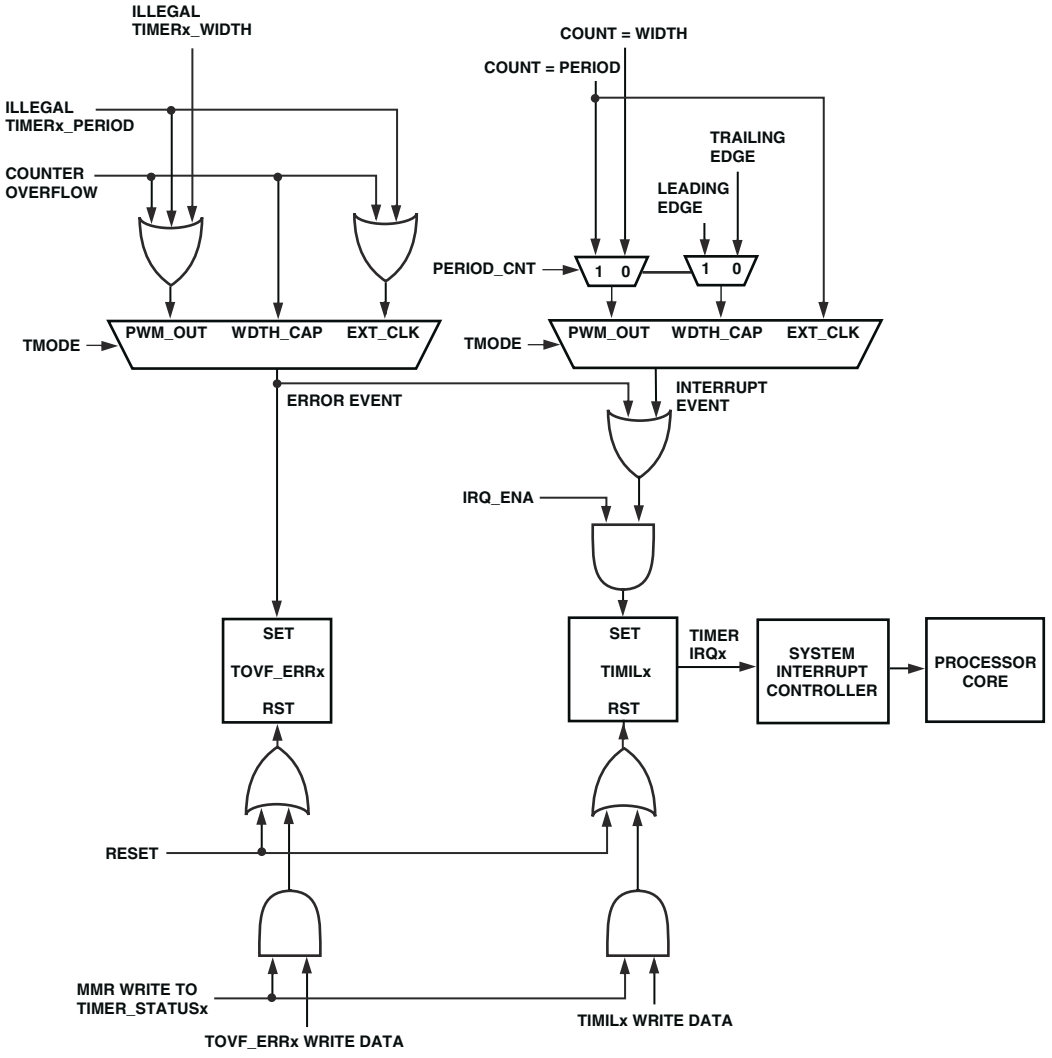


Figure 10-3. Timers Interrupt Structure

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width versus period plausibility checks. Errors are reported by the TOVF_ERRx bits in the TIMER_STATUSx register and the ERR_TYP bit field in the individual TIMERx_CONFIG registers. Table 10-1 provides a summary of error conditions, by using these terms:

- **Startup** The first clock period when the timer counter is running after the timer is enabled by writing TIMER_ENABLEx register.
- **Rollover** The time when the current count matches the value in TIMERx_PERIOD register and the counter is reloaded with the value 1.
- **Overflow** The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged** No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there is no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads 0 if there is no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads 1.

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write-1-to-clear TOVF_ERR to acknowledge the error.

Table 10-1 can be read as: “In mode __ at event __, if TIMERx_PERIOD is __ and TIMERx_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __.”

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMRx pin.

Table 10-1. Overview of Illegal States

Mode	Event	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 10-1. Overview of Illegal States (Cont'd)

Mode	Event	TIMERx_PERIOD	TIMERx_WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_CNT = 0	Startup	Anything	== 0	b#01	Set
		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 10-4 illustrates PWM_OUT mode.

Setting the TMODE field to b#01 in the timer configuration (TIMERx_CONFIG) register enables PWM_OUT mode. Here, the timer TMRx pin is an output, but it can be disabled by setting the OUT_DIS bit in the TIMERx_CONFIG register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

Once a timer is enabled, TIMERx_COUNTER register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the TIMERx_PERIOD register. For either setting of CLK_SEL, when the timer counter equals the timer period, the timer counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMERx_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMERx_PERIOD and TIMERx_WIDTH registers and

generates a repeating (and possibly modulated) waveform. It generates an 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.

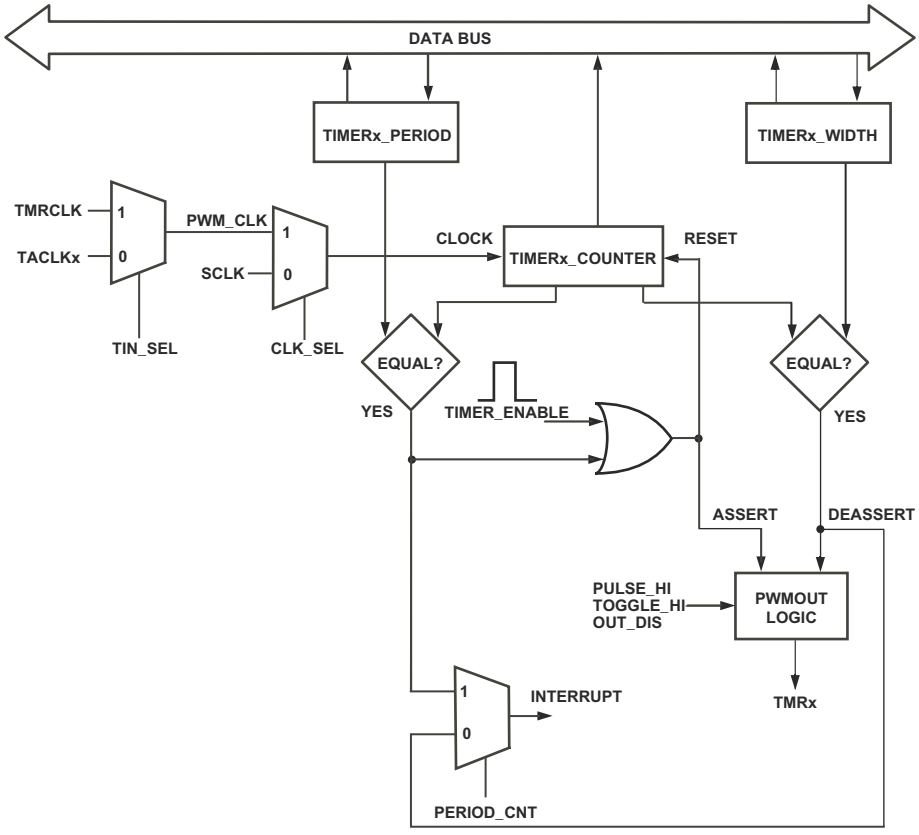


Figure 10-4. Timer Flow Diagram, PWM_OUT Mode

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 TIMERx_CONFIG register. The TMRx pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used. The TMRx pin can also be disabled by the PORTx_FER and the PORTx_MUX registers.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMRx pin. This mode can also be used to implement a precise delay. The pulse width is defined by the TIMERx_WIDTH register, and the TIMERx_PERIOD register is not used. See Figure 10-5.

At the end of the pulse, the timer interrupt latch bit TIMILx is set, and the timer is stopped automatically. No writes to the TIMER_DISABLEx register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMRx pin. If the PULSE_HI bit is not set, the pulse is active low.

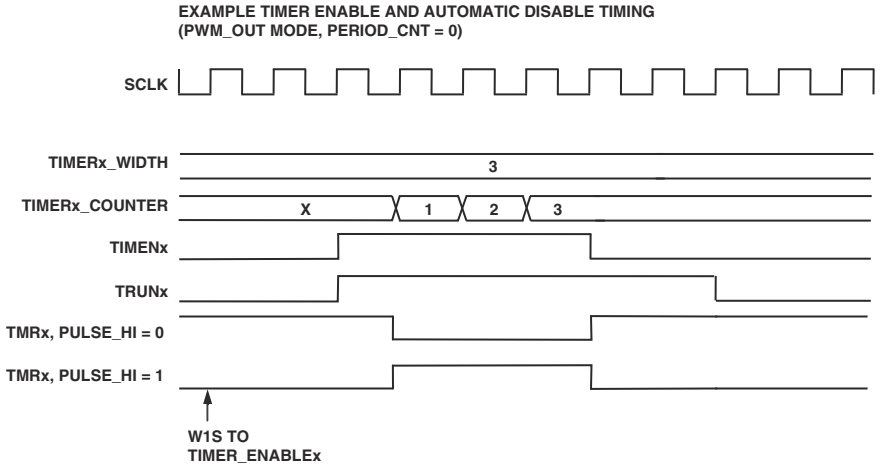


Figure 10-5. Timer Enable and Automatic Disable Timing

The pulse width may be programmed to any value from 1 to (2^{32}-1), inclusive.

Pulse-Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally-clocked timer generates rectangular signals with well-defined period and duty cycles (PWM patterns). This mode also generates periodic interrupts for real-time signal processing.

The 32-bit timer period (TIMERx_PERIOD) and timer pulse width (TIMERx_WIDTH) registers are programmed with the values required by the PWM signal.

When the timer is enabled in this mode, the TMRx pin is pulled to a deasserted state each time the counter equals the value of the TIMERx_WIDTH register. The pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMRx pin, the PULSE_HI bit in the corresponding TIMERx_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMRx pin is driven to the deasserted level.

Figure 10-6 shows timing details.

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine (ISR) must clear the interrupt latch bit (TIMILx) and might alter period and/or width values. In pulse-width modulation (PWM) applications, the software needs to update period and pulse width values while the timer is running. When software updates either period or pulse width registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. Reads from TIMERx_PERIOD and TIMERx_WIDTH return the old values until the period expires.

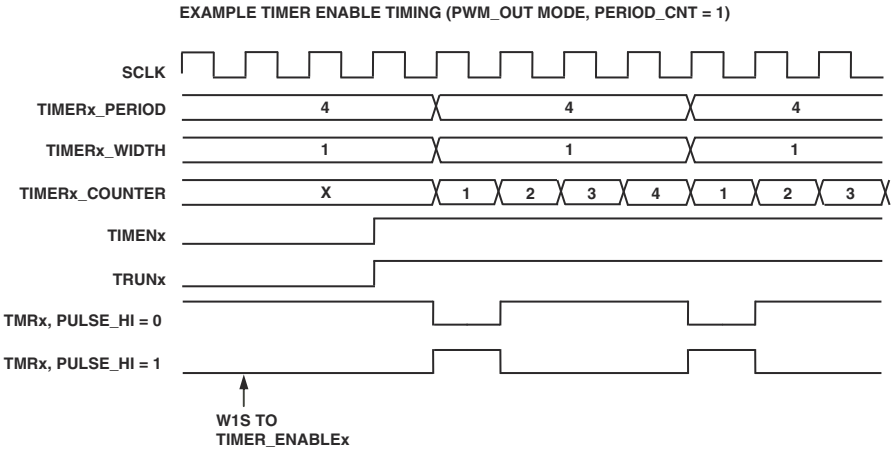


Figure 10-6. Timer Enable Timing

The TOVF_ERRx status bit signifies an error condition in PWM_OUT mode. The TOVF_ERRx bit is set if TIMERx_PERIOD = 0 or TIMERx_PERIOD = 1 at startup, or when TIMERx_COUNTER rolls over. It is also set if the TIMERx_WIDTH register value is greater than or equal to the TIMERx_PERIOD register value by the time the counter rolls over. The ERR_TYP bits are set when the TOVF_ERRx bit is set.

Although the hardware reports an error if the TIMERx_WIDTH value equals the TIMERx_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the TOVL_ERRx flags. Pulse width values greater than the period value are not recommended. Similarly, TIMERx_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

To generate the maximum frequency on the TMRx output pin, set the period value to 2 and the pulse width to 1. This makes TMRx toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (period – 1), inclusive.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (through the TIMERx_WIDTH register). When two timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 10-7.

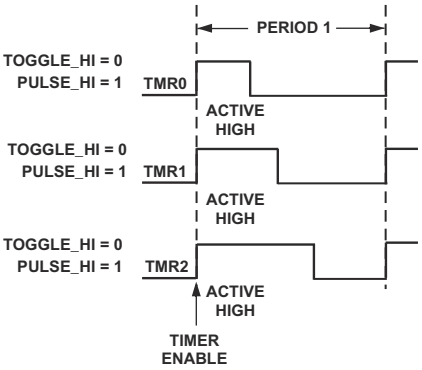


Figure 10-7. Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a fully arbitrary rectangular waveform. The effective waveform is still active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of the TOGGLE_HI bit has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When

PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 10-8 shows an example with three timers running with the same period settings. When software does not alter the PWM settings at run-time, the duty cycle is 50%. The values of the TIMERx_WIDTH registers control the phase between the signals.

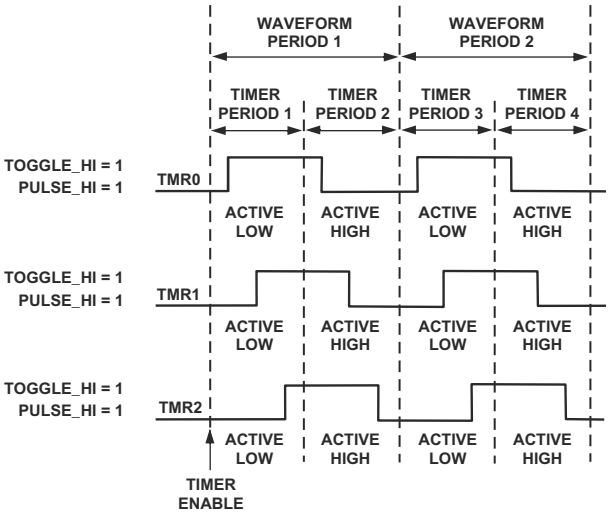


Figure 10-8. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (see Figure 10-9).

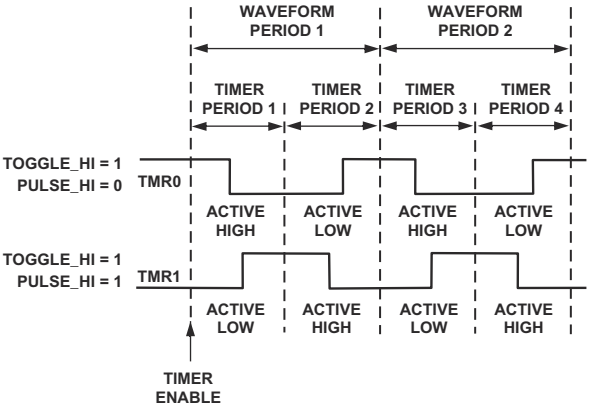


Figure 10-9. Two Timers With Non-Overlapping Clocks

When TOGGLE_HI = 0, software updates the TIMERx_PERIOD and TIMERx_WIDTH registers once per waveform period. When TOGGLE_HI = 1, software updates the TIMERx_PERIOD and TIMERx_WIDTH registers twice per waveform. Period values are half as large. In odd-numbered periods, write (period–width) instead of width to TIMERx_WIDTH in order to obtain center-aligned pulses.

For example, if the pseudo-code when TOGGLE_HI = 0 is:

```
int period, width ;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, period) ;
     write(TIMERx_WIDTH, width) ;
}
```

Then when TOGGLE_HI = 1, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     per1 = period/2 ;
     wid1 = width/2 ;

     per2 = period/2 ;
     wid2 = width/2 ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per1) ;
     write(TIMERx_WIDTH, per1 - wid1) ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per2) ;
     write(TIMERx_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (wid1 does not need to equal wid2). The period can be offset to adjust the phase of the pulses produced (per1 does not need to equal per2).

The timer enable latch (TRUNx bit in the TIMER_STATUSx register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLEx 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 also be taken from the common TMRCLK pin. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse-width modulation waveforms or generates a single pulse with pulse width defined by the 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 PORTx_MUX and PORTx_FER registers.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLEx) as a “stop is pending” condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMRx pin. The processor can determine when the timer stops running by polling for the corresponding TRUNx bit in the TIMER_STATUSx register to read 0 or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMERx_CONFIG cannot be written to a new value) until after the timer stops and TRUNx reads 0.

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLEx to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLEx (and TIMER_DISABLEx) are cleared, and the corresponding TRUNx bit is cleared (See Figure 10-5 on page 10-15). To generate multiple pulses, write a 1 to TIMER_ENABLEx, wait for the timer to stop, then write another 1 to TIMER_ENABLEx.

In continuous PWM generation mode (PWM_OUT, PERIOD_CNT = 1) software can stop the timer by writing to the TIMER_DISABLEx register. To prevent the ongoing PWM pattern from being spoiled in unpredictable fashion, the timer does not stop immediately when the corresponding 1 is written to the TIMER_DISABLEx register. Rather, the write simply clears the enable latch and the timer still completes the ongoing PWM patterns gracefully. It stops cleanly at the end of the first period when the enable latch is cleared. During this final period the TIMENx bit returns 0, but the TRUNx bit still reads as a 1.

If the TRUNx bit is not cleared explicitly, and the enable latch can be cleared and re-enabled all before the end of the current period, the TRUNx bit will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself.

Figure 10-10 shows detailed timing.

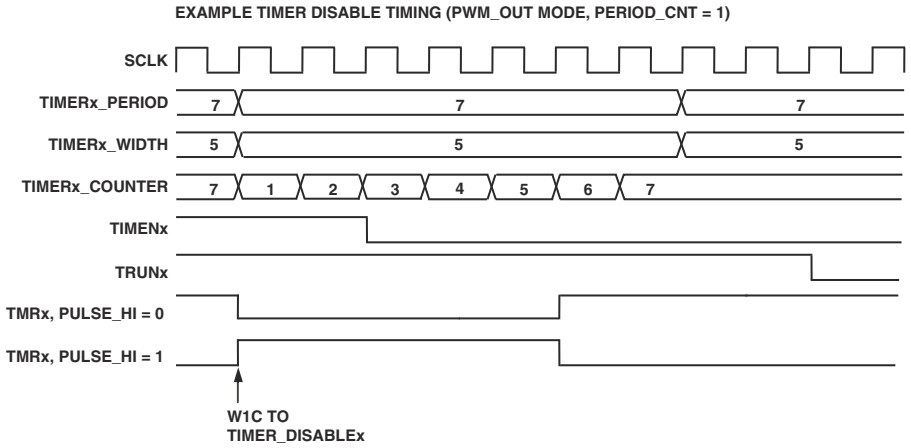


Figure 10-10. Timer Disable Timing

If necessary, the processor can force a timer in PWM_OUT mode to abort immediately. Do this by first writing a 1 to the corresponding bit in TIMER_DISABLEx, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUSx. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMRx pin.

When timers are disabled, the TIMERx_COUNTER registers retain their state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The TIMERx_COUNTER registers are read-only. Software cannot overwrite or preset the timer counter value directly.

Pulse-Width Count and Capture (WDTH_CAP) Mode

Use the WDTH_CAP mode, often simply called “capture mode,” to measure pulse widths on the TMRx or TACIx input pins, or to “receive” PWM signals. Figure 10-11 shows a flow diagram for WDTH_CAP mode.

In WDTH_CAP mode, the TMRx pin is an input pin. The internally-clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the TIMERx_CONFIG register enables this mode.

When enabled in this mode, the timer resets the count in the TIMERx_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMRx pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMERx_COUNTER register into the width buffer register. At the next leading edge, the timer transfers the current 32-bit value of the TIMERx_COUNTER register into the period buffer register. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the TMRx pin, the PULSE_HI bit in the TIMERx_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the content of the TIMERx_COUNTER is captured to the pulse width buffer on the rising edge, and to the period buffer on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the counter value is captured to the pulse width buffer on the falling edge, and to the period buffer on the next rising edge.

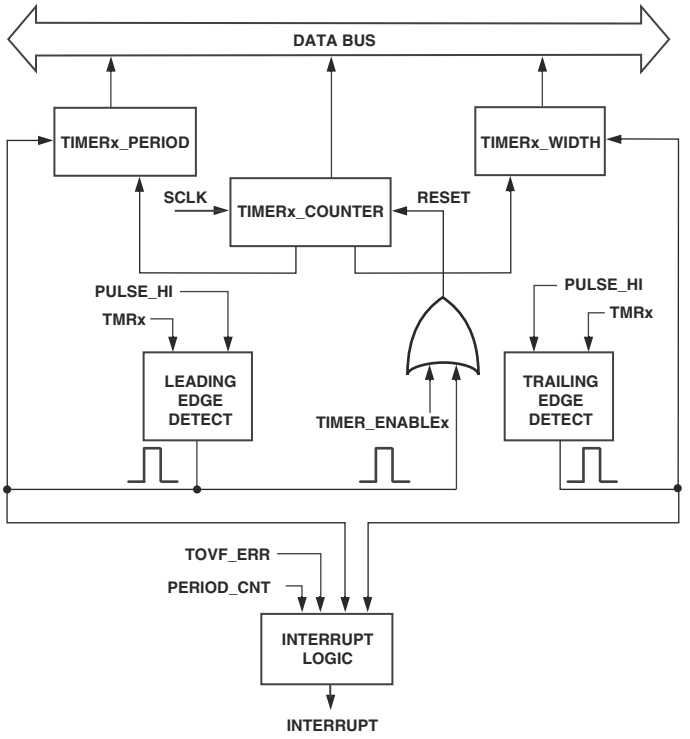


Figure 10-11. Timer Flow Diagram, WDTH_CAP Mode

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 when this set of transactions is executed. Taken together, these three events are called a measurement report. The TOVF_ERRx bit does not get set at a measurement report. A measurement report occurs once per input signal period (at most).

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMERx_PERIOD and TIMERx_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer register captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer register captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (see Figure 10-12), then the TIMERx_PERIOD and TIMERx_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (see Figure 10-13), then the TIMERx_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMERx_PERIOD register reports the pulse period measured at the end of the previous period.

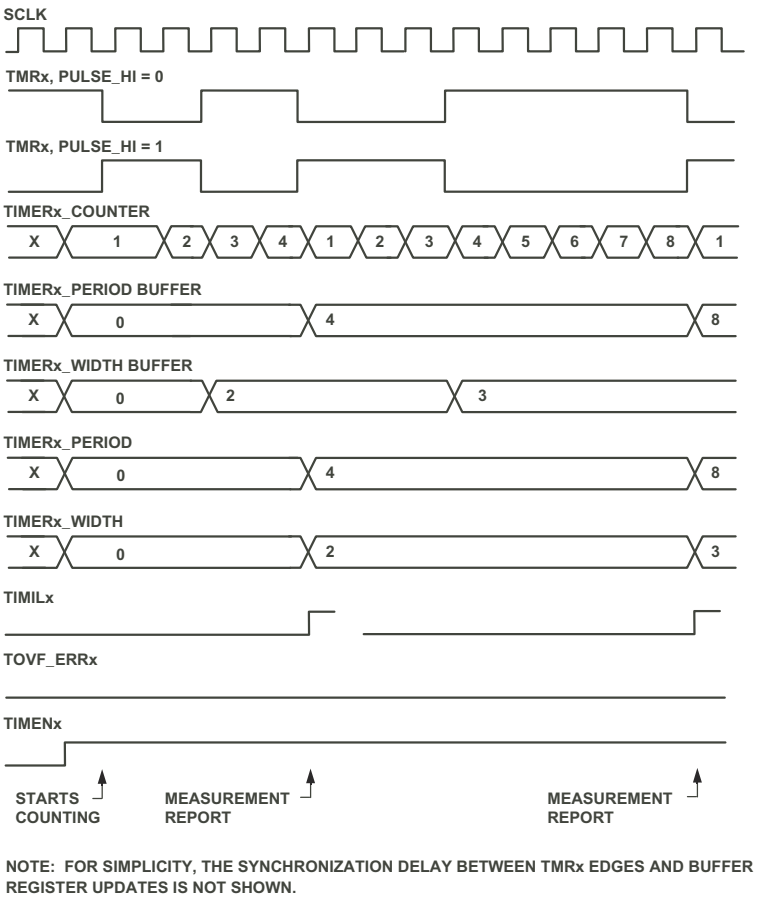
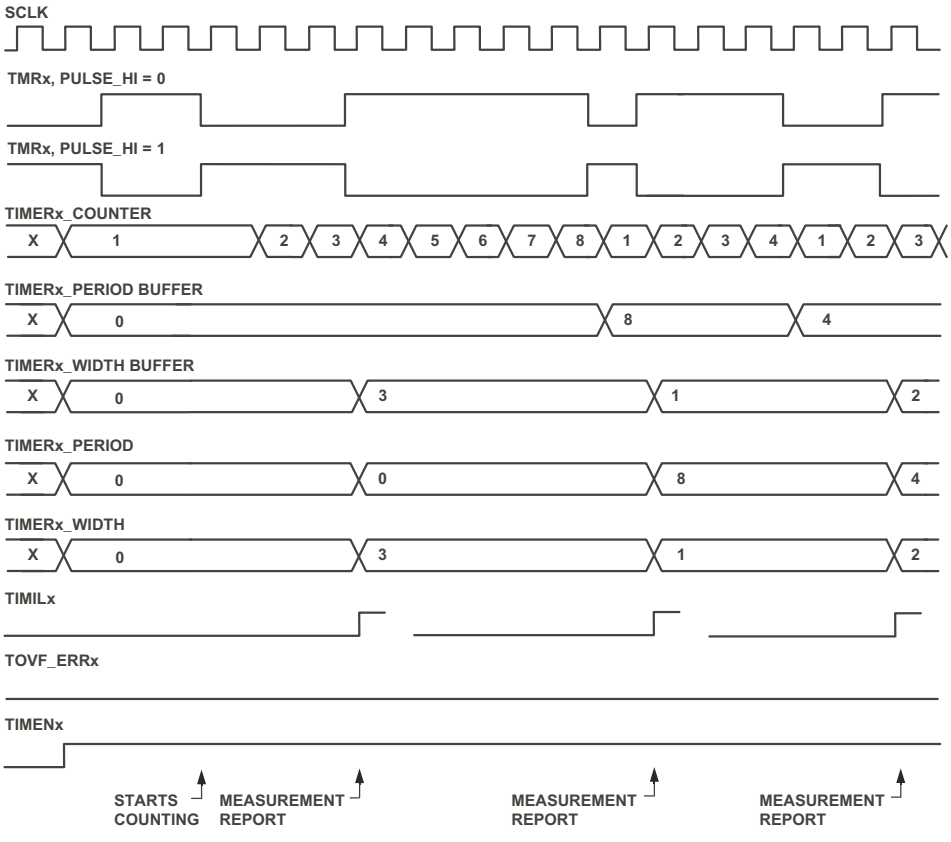


Figure 10-12. Example of Period Capture Measurement Report Timing (WDTH_CAP Mode, PERIOD_CNT = 1)



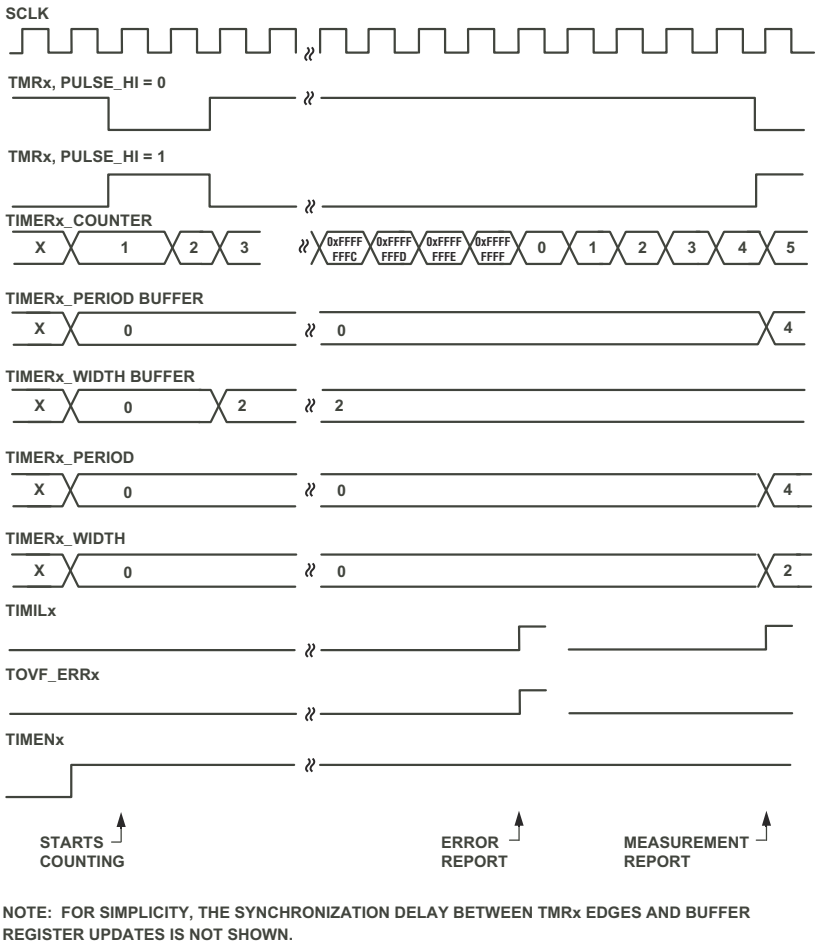
NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-13. Example of Width Capture Measurement Report Timing (WDTH_CAP Mode, PERIOD_CNT = 0)

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMERx_PERIOD value in this case returns 0, as shown in Figure 10-13. To measure the pulse width of a waveform that has only one leading edge and one trailing edge, set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer register. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both TIMERx_WIDTH and TIMERx_PERIOD read 0 (because no measurement report occurred to copy the value captured in the width buffer register to TIMERx_WIDTH). See the first interrupt in Figure 10-14.

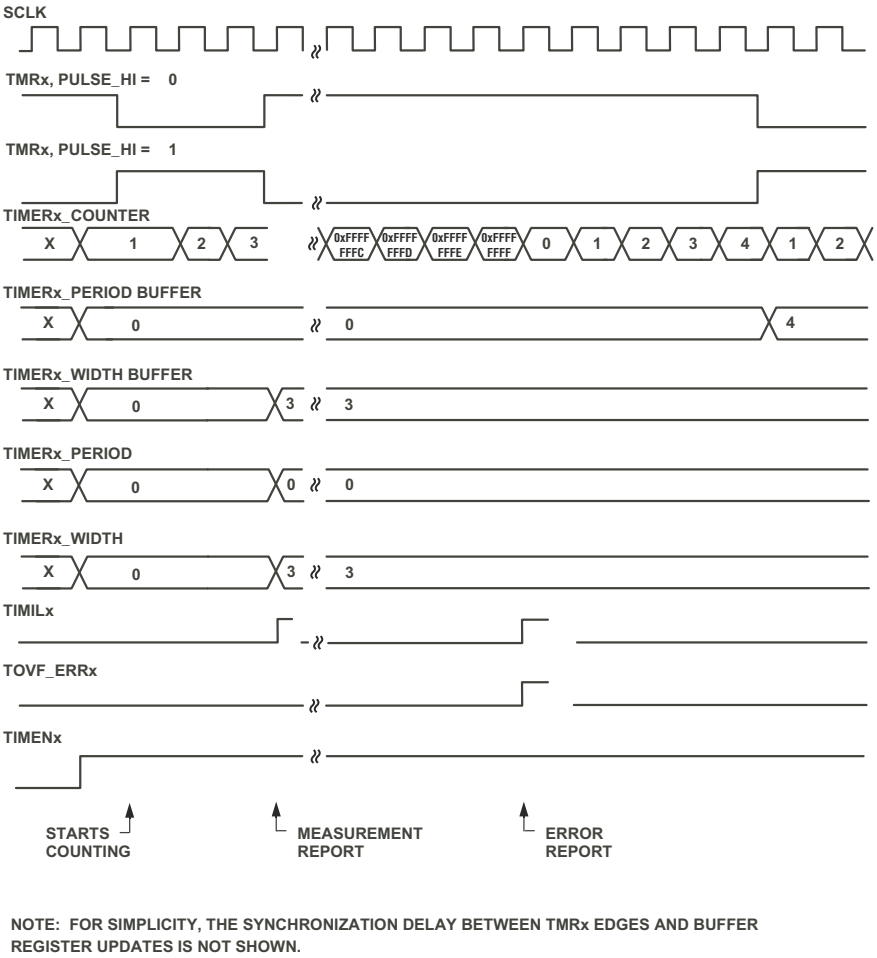
When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement and logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if TIMERx_COUNTER wraps around from 0xFFFF FFFF to 0 in the absence of a leading edge. At that point, the TOVF_ERRx bit in the TIMER_STATUSx register and the ERR_TYP bits in the TIMERx_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMERx_PERIOD and TIMERx_WIDTH registers are never updated at the time an error is signaled. Refer to Figure 10-14 and Figure 10-15 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-14. Example Timing for Period Overflow Followed by Period Capture (WDTH_CAP Mode, PERIOD_CNT = 1)



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-15. Example Timing for Width Capture Followed by Period Overflow (WDTH_CAP Mode, PERIOD_CNT = 0)

Both TIMILx and TOVF_ERRx are sticky bits, and software has to explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMERx_PERIOD nor the TIMERx_WIDTH register were updated. If the timer overflowed and PERIOD_CNT = 0, the TIMERx_PERIOD and TIMERx_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 10-14 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure period = 2 and pulse width = 1.

Autobaud Mode

In WDTH_CAP mode, some of the timers can provide autobaud detection for the universal asynchronous receiver/transmitter (UART) and controller area network (CAN) interfaces. The timer input select (TIN_SEL) bit in the TIMERx_CONFIG register causes the timer to sample the TACIx pin instead of the TMRx pin, when enabled for WDTH_CAP mode. Autobaud detection can be used for initial bit rate negotiations as well as for detection of bit rate drifts while the interface is in operation. For more information on the UART interface, see Chapter 25, “UART Port Controllers”. For more information on the CAN interface, see Chapter 31, “CAN Module”.

Capturing Timings from the GP Counter Module

In WDTH_CAP mode, one of the timers can sense to an internal signal of the GP counter module through the TACI6 input. This enables the timer to capture the period between counter events. For details, see "Capturing Timing Information (Using the General-Purpose Timer)" on page 13-18.

External Event (EXT_CLK) Mode

Use the EXT_CLK mode, sometimes referred to as the "counter mode," to count external events, that is, signal edges on the TMRx pin which is an input in this mode. Figure 10-16 shows a flow diagram for EXT_CLK mode.

The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMERx_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMERx_CONFIG register enables this mode. The TIMERx_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period, and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

After the timer is enabled, it resets TIMERx_COUNTER to 0x0 and then waits for the first leading edge on the TMRx pin. This edge causes TIMERx_COUNTER to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMILx bit is set, and an interrupt is generated. The next leading edge reloads TIMERx_COUNTER again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

The configuration bits, TIN_SEL and PERIOD_CNT, have no effect in this mode. The TOVF_ERRx and ERR_TYP bits are set if TIMERx_COUNTER wraps around from 0xFFFF FFFF to 0 or if period = 0 at startup, or when TIMERx_COUNTER rolls over (from count = period to count = 0x1). TIMERx_WIDTH is unused.

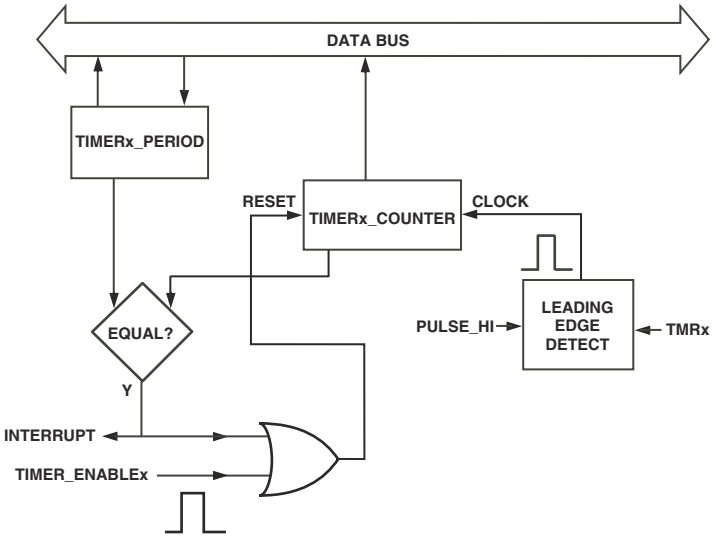


Figure 10-16. Timer Flow Diagram, EXT_CLK Mode

Programming Model

The architecture of the timer block enables any timer to work individually or synchronously along with others in its group. That is, timers 0-7 are members of the same group, and timers 8-10 are members of a separate group. Regardless of the operation mode, the timers' programming model is always straightforward. Because of the error-checking mechanism, always follow this order when enabling timers:

1. Set timer mode.
2. Write TIMERx_WIDTH and TIMERx_PERIOD registers as applicable.
3. Enable timer.

If this order is not followed, the plausibility check may fail because of undefined width and period values, or writes to TIMERx_WIDTH and TIMERx_PERIOD may result in an error condition, because the registers are read-only in some modes. Accordingly, the timer may not start as expected.

If in PWM_OUT mode the PWM patterns of the second period differ from the patterns of the first one, the initialization sequence above might become:

1. Set timer mode to PWM_OUT.
2. Write first TIMERx_WIDTH and TIMERx_PERIOD value pair.
3. Enable timer.
4. Immediately write second TIMERx_WIDTH and TIMERx_PERIOD value pair.

Hardware ensures that the buffered width and period values become active when the first period expires.

Once started, timers require minimal interaction with software, which is usually performed by an interrupt service routine. In PWM_OUT mode software must update the pulse width and/or settings as required. In 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 11 identical timer units.

Each timer has four registers:
"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44

Additionally, three register sets are shared between the 11 timers:
"Timer Enable (TIMER_ENABLEx) Registers" on page 10-38
"Timer Disable (TIMER_DISABLEx) Registers" on page 10-39
"Timer Status (TIMER_STATUSx) Registers" on page 10-40

`TIMER_ENABLE0`, `TIMER_DISABLE0`, and `TIMER_STATUS0` control timers 0 to 7.

`TIMER_ENABLE1`, `TIMER_DISABLE1`, and `TIMER_STATUS1` control timers 8 to 10.

The size of accesses is enforced. A 32-bit access to a `TIMERx_CONFIG` register or a 16-bit access to a `TIMERx_WIDTH`, `TIMERx_PERIOD`, or `TIMERx_COUNTER` register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the `TIMER_ENABLEx`, `TIMER_DISABLEx`, and `TIMER_STATUSx` registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

Table 10-6 on page 10-51 summarizes control bit and register usage in each timer mode.

Timer Enable (TIMER_ENABLEx) Registers

The TIMER_ENABLEx registers, shown in Figure 10-17 and Figure 10-18, allow all timers within a group to be enabled simultaneously in order to make them run completely synchronously. For each timer there is a single W1S control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The bits can be set individually or in any combination. A read of the TIMER_ENABLEx register shows the status of the enable for the corresponding timers within a group. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

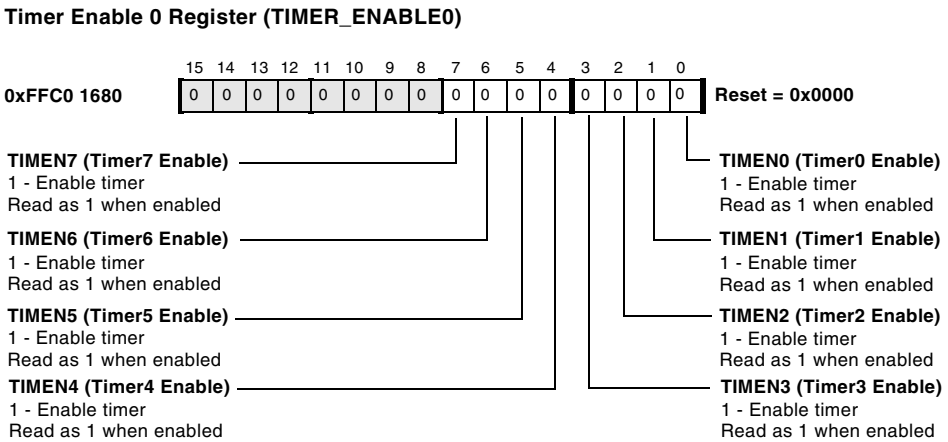


Figure 10-17. Timer Enable 0 Register

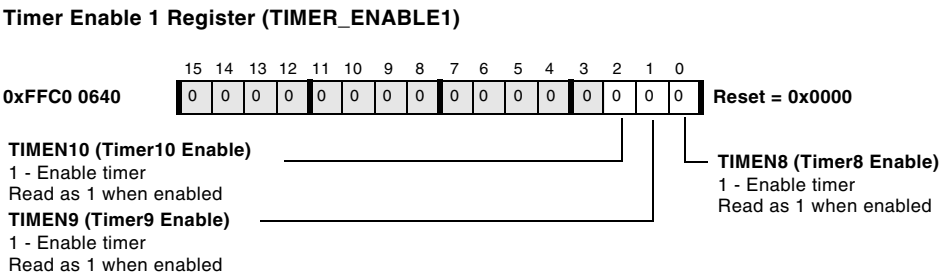


Figure 10-18. Timer Enable 1 Register

Timer Disable (TIMER_DISABLEx) Registers

The TIMER_DISABLEx registers, shown in Figure 10-19 and Figure 10-20 allow all timers within a group to be disabled simultaneously. For each timer there is a single W1C control bit. Writing a 1 disables the corresponding timer; writing a 0 has no effect. The bits within a disable register can be cleared individually or in any combination. A read of the TIMER_DISABLEx register returns a value identical to a read of the corresponding TIMER_ENABLEx register. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

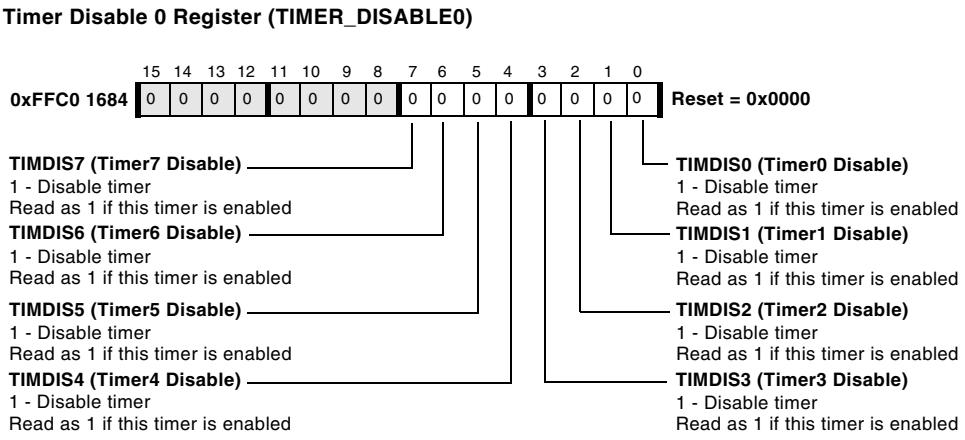


Figure 10-19. Timer Disable 0 Register

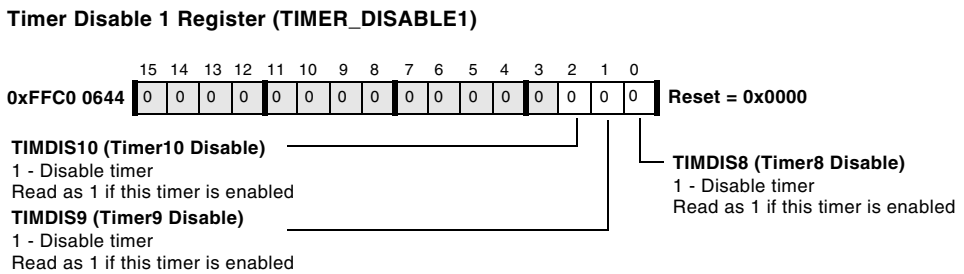


Figure 10-20. Timer Disable 1 Register

In PWM_OUT mode, a write of a 1 to TIMER_DISABLEx does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a 1 to the corresponding bit in TIMER_DISABLEx, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUSx. See "Stopping the Timer in PWM_OUT Mode" on page 10-23.

In WDTH_CAP and EXT_CLK modes, a write of a 1 to TIMER_DISABLEx stops the corresponding timer immediately.

Timer Status (TIMER_STATUSx) Registers

The TIMER_STATUSx registers are used to check the status of all timers within a group with a single read (see Figure 10-21 and Figure 10-22). Status bits are sticky and W1C. The TRUNx bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a TIMER_STATUSx register read access, all reserved or unused bits return a 0.

For detailed behavior and usage of the TRUNx bit see "Stopping the Timer in PWM_OUT Mode" on page 10-23. 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 10-10.

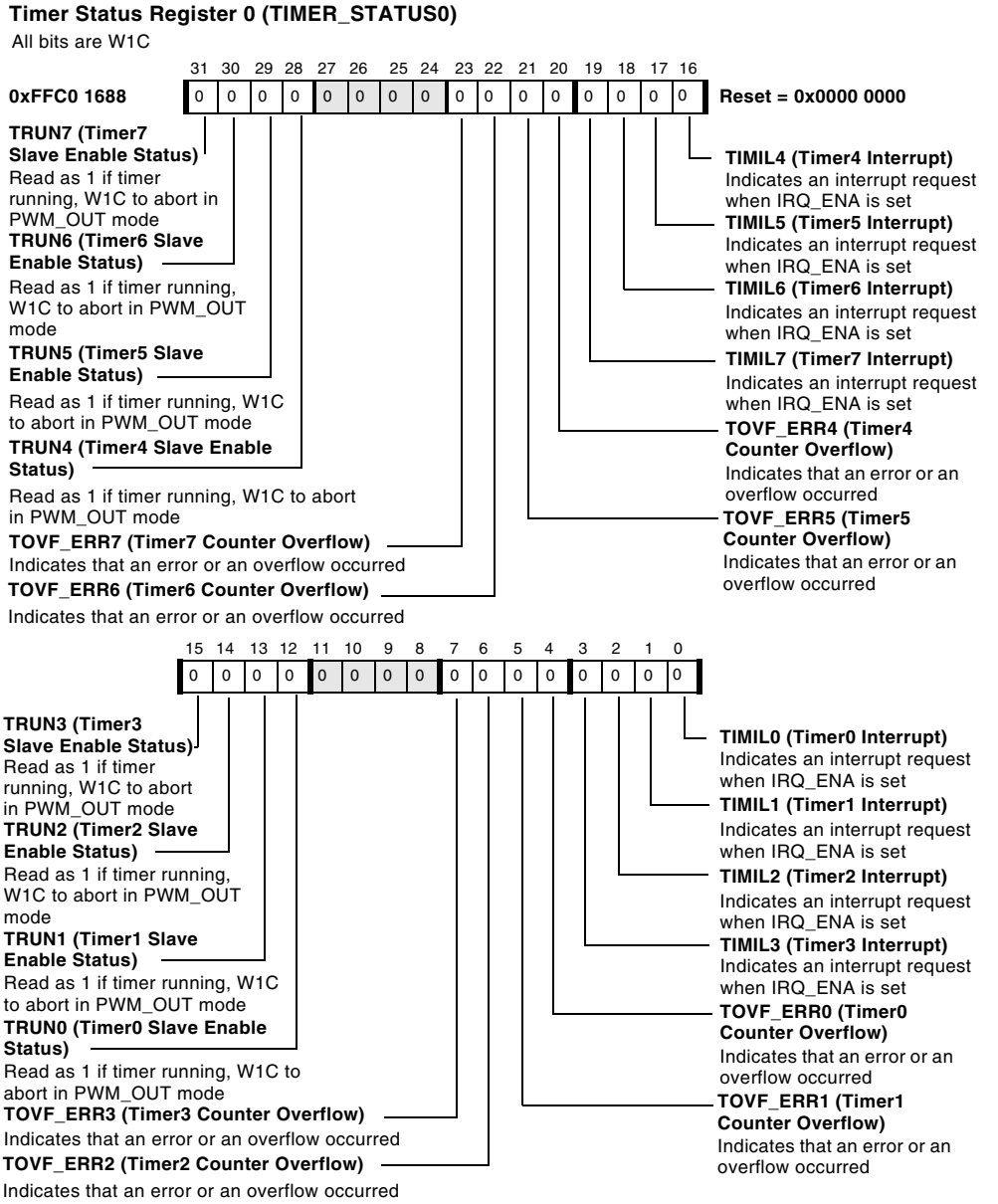


Figure 10-21. Timer Status 0 Register

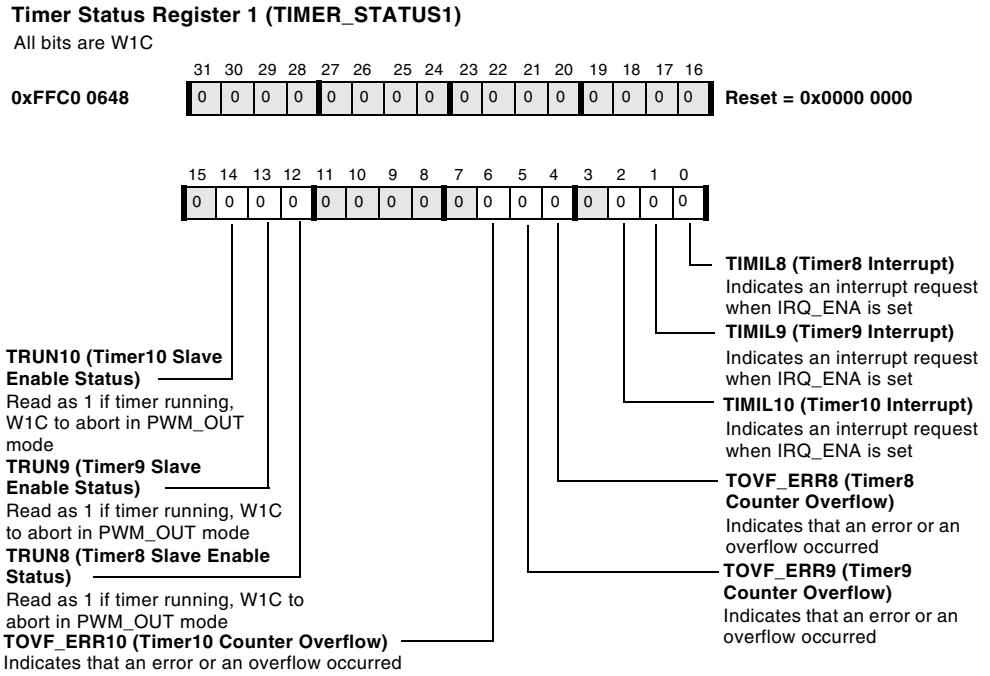


Figure 10-22. Timer Status 1 Register

Timer Configuration (TIMERx_CONFIG) Registers

Each timer's operating mode is specified by its TIMERx_CONFIG register (Figure 10-23 and Table 10-2), which may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUNx bit in TIMER_STATUSx before attempting to reprogram TIMERx_CONFIG. The TIMERx_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled. Each time TOVF_ERRx is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 10-1

on page 10-11. 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.

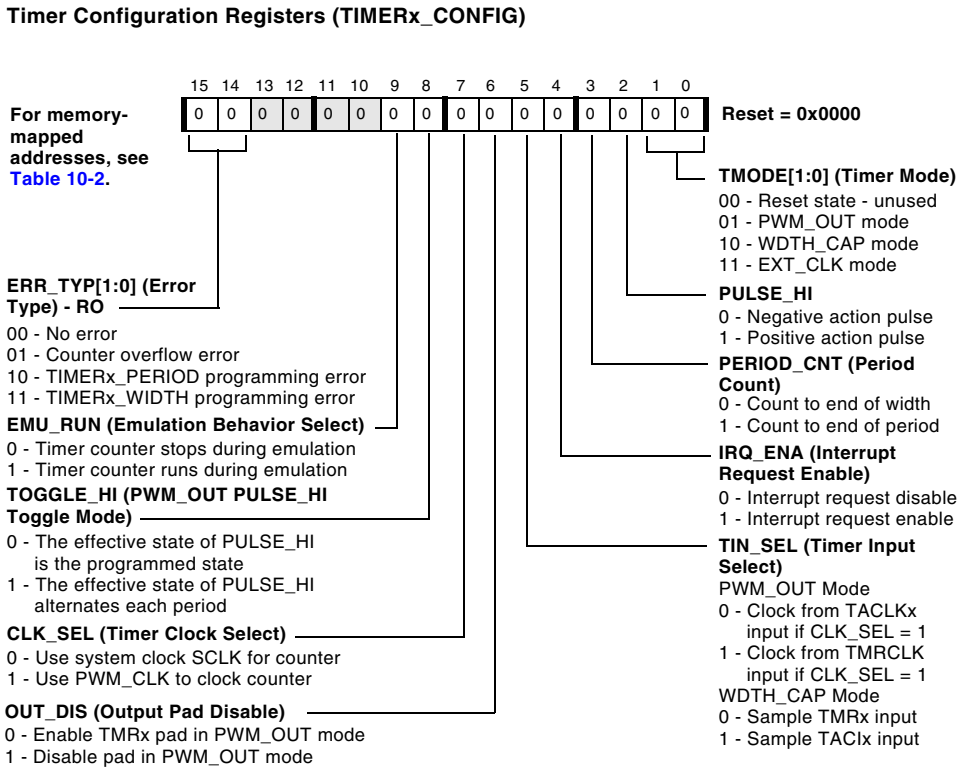


Figure 10-23. Timer Configuration Registers

Table 10-2. Timer Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_CONFIG	0xFFC0 1600
TIMER1_CONFIG	0xFFC0 1610
TIMER2_CONFIG	0xFFC0 1620
TIMER3_CONFIG	0xFFC0 1630
TIMER4_CONFIG	0xFFC0 1640
TIMER5_CONFIG	0xFFC0 1650
TIMER6_CONFIG	0xFFC0 1660
TIMER7_CONFIG	0xFFC0 1670
TIMER8_CONFIG	0xFFC0 0600
TIMER9_CONFIG	0xFFC0 0610
TIMER10_CONFIG	0xFFC0 0620

Timer Counter (TIMERx_COUNTER) Registers

These read-only registers retain their state when disabled. When enabled, the TIMERx_COUNTER register is reinitialized by hardware based on configuration and mode. The TIMERx_COUNTER register, shown in Figure 10-24 and Table 10-3, may be read at any time (whether the timer is running or stopped), and it returns an atomic 32-bit value. Depending on the operation mode, the incrementing counter can be clocked by four different sources: SCLK, the TMRx pin, the alternative timer clock pin TACLKx, or the common TMRCLK pin, which is most likely used as the EPPI0 clock (EPPI0_CLK).

When the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMERx_COUNTER also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMRx pin waveform is “stretched”; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on TMRx may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMERx_PERIOD and TIMERx_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMERx_CONFIG to enable this behavior.

Timer Counter Registers (TIMERx_COUNTER)

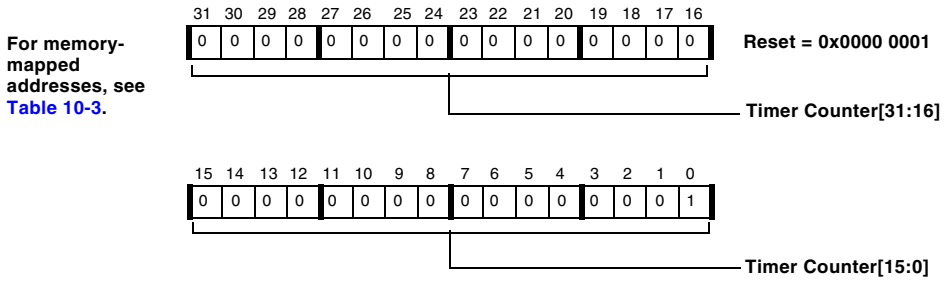


Figure 10-24. Timer Counter Registers

Table 10-3. Timer Counter Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_COUNTER	0xFFC0 1604
TIMER1_COUNTER	0xFFC0 1614
TIMER2_COUNTER	0xFFC0 1624
TIMER3_COUNTER	0xFFC0 1634
TIMER4_COUNTER	0xFFC0 1644
TIMER5_COUNTER	0xFFC0 1654
TIMER6_COUNTER	0xFFC0 1664
TIMER7_COUNTER	0xFFC0 1674
TIMER8_COUNTER	0xFFC0 0604
TIMER9_COUNTER	0xFFC0 0614
TIMER10_COUNTER	0xFFC0 0624

TIMERx_PERIOD and TIMERx_WIDTH Registers

Usage of the TIMERx_PERIOD register, shown in Figure 10-25 and Table 10-3, and the TIMERx_WIDTH register, shown in Figure 10-26 and Table 10-4, varies depending on the mode of the timer:

- In pulse width modulation mode (PWM_OUT), both the TIMERx_PERIOD and TIMERx_WIDTH register values can be updated "on-the-fly" since these values change simultaneously.
- In pulse-width and period capture mode (WDTH_CAP), the timer period and timer pulse width buffer values are captured at the appropriate time. The TIMERx_PERIOD and TIMERx_WIDTH registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In external event capture mode (EXT_CLK), the TIMERx_PERIOD is writable and can be updated "on-the-fly." TIMERx_WIDTH is not used.

When a timer is enabled and running, and the software writes new values to TIMERx_PERIOD and TIMERx_WIDTH, the writes are buffered and do not update the registers until the end of the current period (when the value in TIMERx_COUNTER equals the value in TIMERx_PERIOD).

If new values are not written to TIMERx_PERIOD or TIMERx_WIDTH, the value from the previous period is reused. Writes to the 32-bit TIMERx_PERIOD and TIMERx_WIDTH registers are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the TIMERx_PERIOD or TIMERx_WIDTH registers are always stored in the buffer registers. Reads from the TIMERx_PERIOD or TIMERx_WIDTH registers always return the current, active value of period or pulse width. Written values are not read back until they become active.

When the timer is enabled, they do not become active until after TIMERx_PERIOD and TIMERx_WIDTH are updated from their respective buffers at the end of the current period. See Figure 10-2 on page 10-4.

When the timer is disabled, writes to the buffer registers are immediately copied through to the TIMERx_PERIOD or TIMERx_WIDTH register so that they are ready for use in the first timer period. For example, to change the values for the TIMERx_PERIOD or TIMERx_WIDTH registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In PWM_OUT mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both TIMERx_PERIOD and TIMERx_WIDTH. The next period may use one old value and one new values.

In order to prevent “pulse width >= period” errors, write TIMERx_WIDTH before TIMERx_PERIOD when decreasing the values, and write TIMERx_PERIOD before TIMERx_WIDTH when increasing the value.

Timer Period Registers (TIMERx_PERIOD)

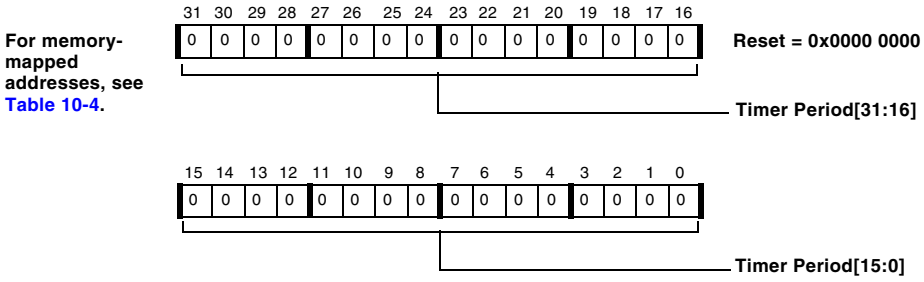


Figure 10-25. Timer Period Registers

Table 10-4. Timer Period Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_PERIOD	0xFFC0 1608
TIMER1_PERIOD	0xFFC0 1618
TIMER2_PERIOD	0xFFC0 1628
TIMER3_PERIOD	0xFFC0 1638
TIMER4_PERIOD	0xFFC0 1648
TIMER5_PERIOD	0xFFC0 1658
TIMER6_PERIOD	0xFFC0 1668
TIMER7_PERIOD	0xFFC0 1678
TIMER8_PERIOD	0xFFC0 0608
TIMER9_PERIOD	0xFFC0 0618
TIMER10_PERIOD	0xFFC0 0628

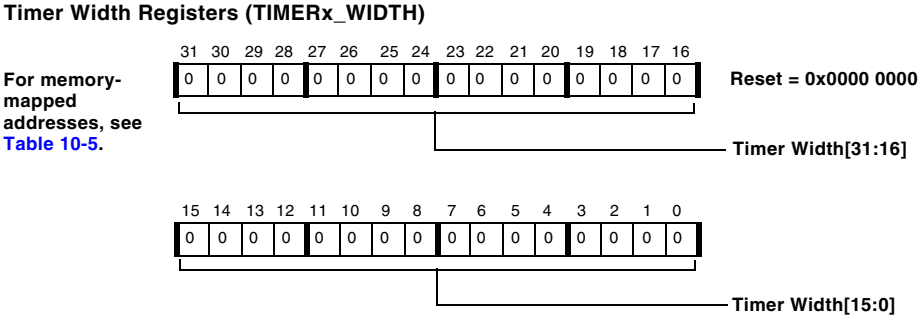


Figure 10-26. Timer Width Registers

Table 10-5. Timer Width Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_WIDTH	0xFFC0 160C
TIMER1_WIDTH	0xFFC0 161C
TIMER2_WIDTH	0xFFC0 162C
TIMER3_WIDTH	0xFFC0 163C
TIMER4_WIDTH	0xFFC0 164C
TIMER5_WIDTH	0xFFC0 165C
TIMER6_WIDTH	0xFFC0 166C
TIMER7_WIDTH	0xFFC0 167C
TIMER8_WIDTH	0xFFC0 060C
TIMER9_WIDTH	0xFFC0 061C
TIMER10_WIDTH	0xFFC0 062C

Summary

Table 10-6 summarizes control bit and register usage in each timer mode.

Table 10-6. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLEx	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLEx	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt
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

Table 10-6. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused
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

Table 10-6. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOVF_ERR	Set at startup or rollover if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or rollover if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Programming Examples

Listing 10-1 configures the `PORTA_FER` register in a way that all eight `TMRx` pins are connected to port A.

Listing 10-1. Port Setup

```
timer_port_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(PORTA_FER);
    p5.l = lo(PORTA_FER);
    r7.l = PA1|PA5;
    w[p5] = r7;
    p5.l = lo(PORTA_MUX);
    r7.l = PFTE;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
```

```
    rts;
timer_port_setup.end:
```

Listing 10-2 generates signals on the TMR4 (PA1) and TMR5 (PA5) outputs. By default, timer 5 generates a continuous PWM signal with a duty cycle of 50% (period = 0x40 SCLKs, width = 0x20 SCLKs) while the PWM signal generated by timer 4 has the same period but 25% duty cycle (width = 0x10 SCLKs).

If the preprocessor constant SINGLE_PULSE is defined, every TMRx pin outputs only a single high pulse of 0x20 (timer 4) and 0x10 SCLKs (timer 5) duration.

In any case, the timers are started synchronously and the rising edges are aligned, that is, the pulses are left-aligned.

Listing 10-2. Signal Generation

```
// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE0] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r7;
    r7 = 0x20 (z);
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r7;
#ifndef SINGLE_PULSE
```

```
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. Listing 10-3 illustrates how interrupts are generated and how interrupt service routines can be registers. In this example, the timer 5 interrupt is assigned to the IVG7 interrupt channel of the CEC controller.

Listing 10-3. Interrupt Setup

```
timer5_interrupt_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(IMASK);
    p5.l = lo(IMASK);
/* register interrupt service routine */
    r7.h = hi(isr_timer5);
    r7.l = lo(isr_timer5);
    [p5 + EVT7 - IMASK] = r7;
/* unmask IVG7 in CEC */
    r7 = [p5];
    bitset(r7, bitpos(EVT_IVG7));
    [p5] = r7;
    p5.h = hi(SIC_IMASK2);
    p5.l = lo(SIC_IMASK2);
/* assign timer 5 IRQ = IRQ91 to IVG7 */
    r7.h = hi(P91_IVG(7));
    r7.l = lo(P91_IVG(7));
    [p5 + SIC_IAR11 - SIC_IMASK2] = r7;
```

```
/* enable timer 5 IRQ */
    r7 = [p5];
    bitset(r7, 27);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:
```

The example shown in Listing 10-4 does not drive the TMRx pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant SINGLE_PULSE is defined, timer 5 requests an interrupt only once. Unlike in a real application, the purpose of the interrupt service routine shown in this example is clearing of the interrupt request and counting interrupt occurrences.

Listing 10-4. Periodic Interrupt Requests

```
// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7 = 0x1000 (z);
#ifndef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r7;
    r7 = 0x1 (z);
#endif
```

```
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r7;
    r7.l = TIMEN5;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    r0 = 0 (z);
    rts;
timer5_interrupt_generation.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_STATUS0);
    p5.l = lo(TIMER_STATUS0);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Figure 10-27 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 10-7 summarizes the register writes.

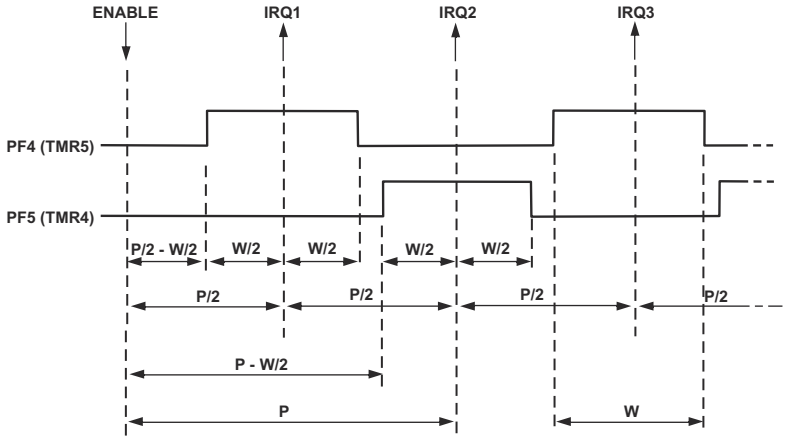


Figure 10-27. Non-Overlapping Clock Pulses

Table 10-7. Register Writes for Non-Overlapping Clock Pulses

Register	Before Enable	After Enable	At IRQ1	At IRQ2
TIMER5_PERIOD	P/2			
TIMER5_WIDTH	P/2 -W/2	W/2	P/2 - W/2	W/2
TIMER4_PERIOD	P	P/2		
TIMER4_WIDTH	P -W/2		W/2	P/2-W-2

Since hardware only updates the written period and width values at the end of periods, software can write new values immediately after the timers have been enabled. Note that both timers' period expires at exactly the same times with the exception of the first timer 5 interrupt (at `IRQ1`) which is not visible to timer 4.

Listing 10-5 illustrates how two timers can generate two non-overlapping clock pulses as typically required for break-before-make scenarios. Both timers are running in `PWM_OUT` mode with `PERIOD_CNT` = 1 and `PULSE_HI` = 1.

Listing 10-5 generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern.

Listing 10-5. Non-Overlapping Clock Pulses

```
#define P 0x1000 /* signal period */
#define W 0x0600 /* signal pulse width */
#define N 4      /* number of pulses before disable */
timer45_toggle_hi:
    [--sp] = (r7:1, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE0] = r7;
/* calculate timers widths and period */
    r0.l = lo(P);
    r0.h = hi(P);
    r1.l = lo(W);
    r1.h = hi(W);
    r2 = r1 >> 1;   /* W/2 */
    r3 = r0 >> 1;   /* P/2 */
    r4 = r3 - r2;   /* P/2 - W/2 */
    r5 = r0 - r2;   /* P - W/2 */
/* write values for initial period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r4;
/* start timers */
    r7.l = TIMEN5 | TIMEN4;
    w[p5 + TIMER_ENABLE0 - TIMER_ENABLE0] = r7;
```

```
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r2;
/* r0 functions as signal period counter */
    r0.h = hi(N * 2 - 1);
    r0.l = lo(N * 2 - 1);
    (r7:1, p5:5) = [sp++];
    rts;
timer45_toggle_hi.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:5, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS0 - TIMER_ENABLE0] = r7;
/* toggle width values (width = period - width) */
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE0];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE0];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE0];
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r7;
/* disable after a certain number of periods */
    r0+= -1;
    CC = r0 == 0;
    r5.l = 0;
    r7.l = TIMDIS5 | TIMDIS4;
    if!CC r7 = r5;
```

```
    w[p5 + TIMER_DISABLE0 - TIMER_ENABLE0] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 10-6 configures timer 5 in WDTH_CAP mode. If looped back externally, this code can be used to receive N PWM patterns generated by one of the other timers. Ensure that the PWM generator uses the same PERIOD_CNT and PULSE_HI settings.

Listing 10-6. Timer Configured in WDTH_CAP Mode

```
.section L1_data_a;
.align 4;
#define N 1024
.var buffReceive[N*2];
.section L1_code;
timer5_capture:
    [--sp] = (r7:7, p5:5);
/* setup DAG2 */
    r7.h = hi(buffReceive);
    r7.l = lo(buffReceive);
    i2 = r7;
    b2 = r7;
    l2 = length(buffReceive)*4;
/* config timer for high pulses capture */
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7.l = TIMEN5;
    w[p5 + TIMER_ENABLE0 - TIMER_ENABLE0] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
```

```
timer5_capture.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
/* clear interrupt request first */
    p5.h = hi(TIMER_STATUS0);
    p5.l = lo(TIMER_STATUS0);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMER0_PERIOD - TIMER_STATUS0];
    [i2++] = r7;
    r7 = [p5 + TIMER0_WIDTH - TIMER_STATUS0];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

11 CORE TIMER

This chapter describes the core timer and includes the following sections:

- "Overview and Features" on page 11-1
- "Timer Overview" on page 11-2
- "Description of Operation" on page 11-3
- "Core Timer Registers" on page 11-4
- "Programming Examples" on page 11-7

Overview and Features

The core timer is a programmable, 32-bit interval timer that can generate periodic interrupts. Unlike other peripherals, the core timer resides inside the Blackfin processor core and runs at the core clock (CCLK) rate.

Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operation at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

The following sections provide an overview of the core timer.

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The module is clocked by the core clock `CCLK`. The timer has its dedicated interrupt request signal which is of higher priority than all other peripherals' requests.

Figure 11-1 provides a block diagram of the core timer.

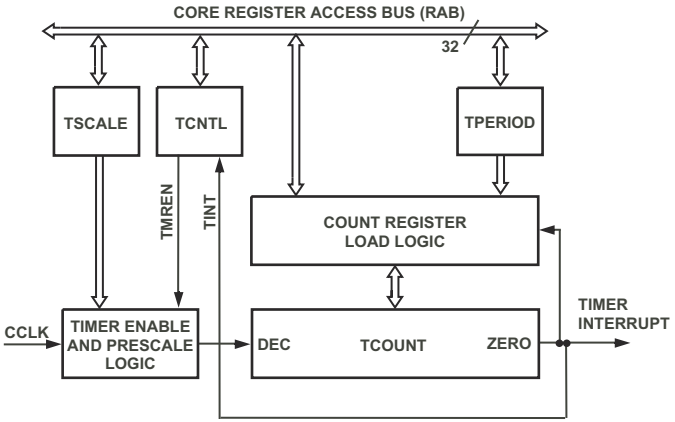


Figure 11-1. Core Timer Block Diagram

Description of Operation

It is up to software to initialize the core timer's counter (TCOUNT) *before* the timer is enabled. The TCOUNT register can be written directly. However, writes to the TPERIOD register are also passed through to the counter, TCOUNT.

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every time the prescaler (TSCALE) expires, that is, every TSCALE + 1 number of CCLK clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register.

If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. If the TAUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR bit is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN bit is set when TMPWR = 0.

Interrupt Processing

The core timer has its dedicated interrupt request signal which is of higher priority than all other peripherals' requests. The requests goes directly to the Core Event Controller (CEC) and does not pass the System Interrupt Controller (SIC). Therefore, the interrupt processing is also completely in the CCLK domain.

Unlike requests from other Blackfin processor peripherals, the core interrupt request is edge sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

The TINT bit in the TCNTL register indicates that an interrupt is generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Core Timer Registers

The core timer includes the following four core memory-mapped registers (MMRs):

- "Core Timer Control (TCNTL) Register" on page 11-5
- "Core Timer Count (TCOUNT) Register" on page 11-5
- "Core Timer Period (TPERIOD) Register" on page 11-6
- "Core Timer Scale (TSCALE) Register" on page 11-7

Similar to all core MMRs, these registers are always accessed by 32-bit read and write operations.

Core Timer Control (TCNTL) Register

The core timer control (TCNTL) register, shown in Figure 11-2, functions as a control and status register.

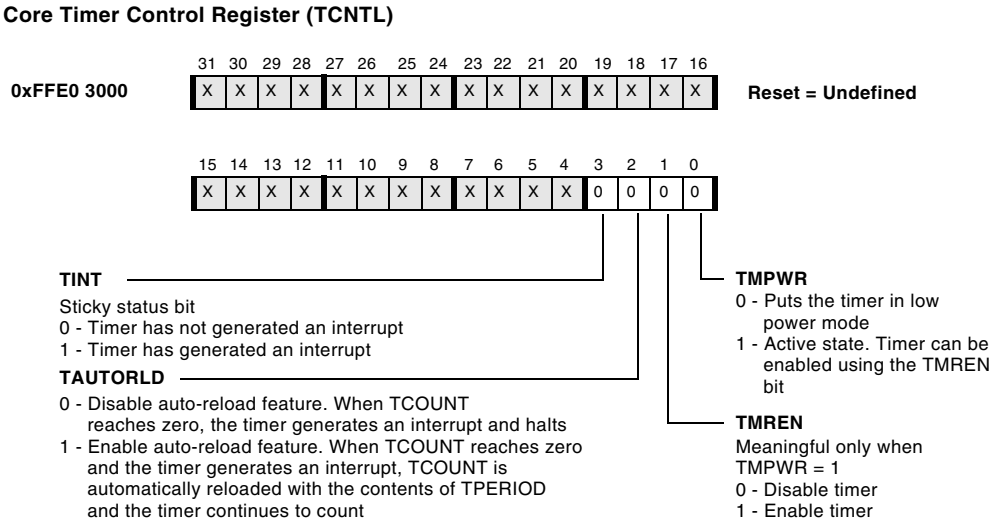


Figure 11-2. Core Timer Control Register

Core Timer Count (TCOUNT) Register

The core timer count register (TCOUNT) shown in Figure 11-3 decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register as well. Nevertheless, the TCOUNT register can be written directly. In auto-reload mode the value written to TCOUNT may differ from the TPERIOD value to let the initial period be shorter or longer than the following ones. To do this, write to TPERIOD first and overwrite TCOUNT register afterward.

Writes to TCOUNT are ignored once the timer is running.

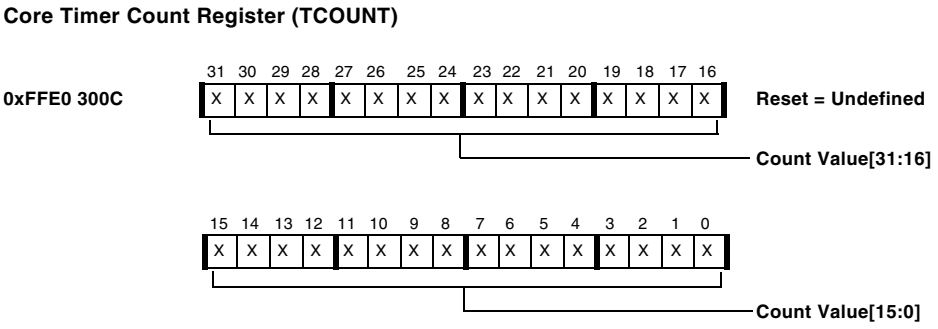


Figure 11-3. Core Timer Count Register

Core Timer Period (TPERIOD) Register

When auto-reload is enabled, the TCOUNT register is reloaded with the value of the core timer period register (TPERIOD, shown in Figure 11-4), whenever TCOUNT register reaches 0. Writes to TPERIOD register are ignored when the timer is running.

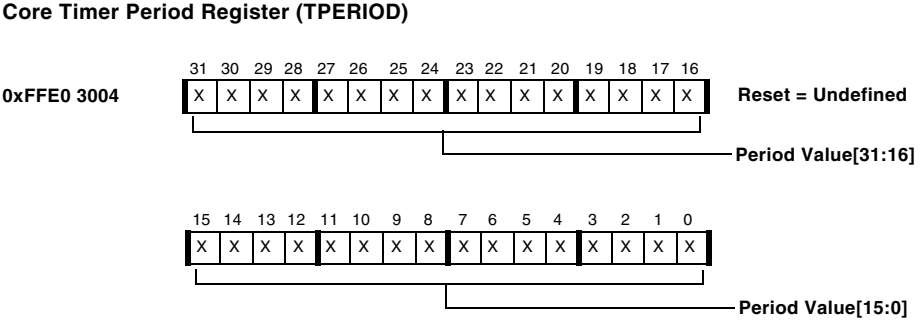


Figure 11-4. Core Timer Period Register

Core Timer Scale (TSCALE) Register

The core timer scale register (TSCALE, shown in Figure 11-5,) stores the scaling value that is one less than the number of cycles between decrements of TCOUNT register. For example, if the value in the TSCALE register is 0, the counter register decrements once every CCLK clock cycle. If the value of TSCALE register is 1, the counter decrements once every two cycles.

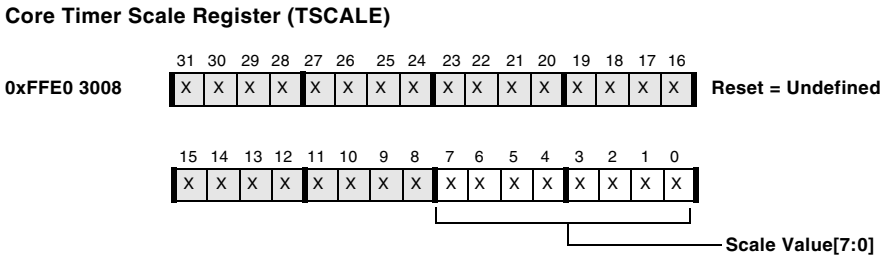


Figure 11-5. Core Timer Scale Register

Programming Examples

Listing 11-1 configures the core timer in auto reload mode. Assuming a CCLK of 500 MHz, the resulting period is 1 s. The initial period is twice as long as the others.

Listing 11-1. Core Timer Configuration

```
#include <blackfin.h>
.section L1_code;
.global _main;
_main:
/* Register service routine at EVT6 and unmask interrupt */
    p1.l = lo(IMASK);
```

```
    p1.h = hi(IMASK);
    r0.l = lo(isr_core_timer);
    r0.h = hi(isr_core_timer);
    [p1 + EVT6 - IMASK] = r0;
    r0 = [p1];
    bitset(r0, bitpos(EVT_IVTMR));
    [p1] = r0;
/* Prescaler = 50, Period = 10,000,000, First Period = 20,000,000
*/
    p1.l = lo(TCNTL);
    p1.h = hi(TCNTL);
    r0 = 50 (z);
    [p1 + TSCALE - TCNTL] = r0;
    r0.l = lo(10000000);
    r0.h = hi(10000000);
    [p1 + TPERIOD - TCNTL] = r0;
    r0 <<= 1;
    [p1 + TCOUNT - TCNTL] = r0;
/* R6 counts interrupts */
    r6 = 0 (z);
/* start in auto-reload mode */
    r0 = TAUTORLD | TMPWR | TMREN (z);
    [p1] = r0;
_main.forever:
    jump _main.forever;
_main.end:
/* interrupt service routine simple increments R6 */
isr_core_timer:
    [--sp] = astat;
    r6+= 1;
    astat = [sp++];
    rti;
isr_core_timer.end:
```

12 WATCHDOG TIMER

This chapter describes the watchdog timer and includes the following sections:

- "Overview and Features" on page 12-1
- "Interface Overview" on page 12-3
- "Description of Operation" on page 12-4
- "Watchdog Timer Registers" on page 12-6
- "Programming Examples" on page 12-9

Overview and Features

The Blackfin processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the watchdog expires before being updated by software.

Watchdog timer key features include:

- 32-bit watchdog timer
- 8-bit disable bit pattern
- System reset on expire option

- NMI on expire option
- General-purpose interrupt option

Typically, the watchdog timer is used to supervise stability of the system software. When used in this way, software reloads the watchdog timer in a regular manner so that the downward counting timer never expires (never becomes 0). An expiring timer then indicates that system software might be out of control. At this point a special error handler may recover the system. For safety, however, it is often better to reset and reboot the system directly by hardware control.

Especially in slave boot configurations, a processor reset cannot automatically force the part to reboot. In this case, the processor may reset without booting again and may negotiate with the host device by the time program execution starts. Alternatively, a watchdog event can cause an NMI event. The NMI service routine may request the host device to reset and/or reboot the Blackfin processor.

Often, the watchdog timer is also programmed to let the processor wake up from sleep mode after a programmable period of time.

For easier debugging, the watchdog timer does not decrement (even if enabled) when the processor is in emulation mode.

Interface Overview

Figure 12-1 provides a block diagram of the watchdog timer.

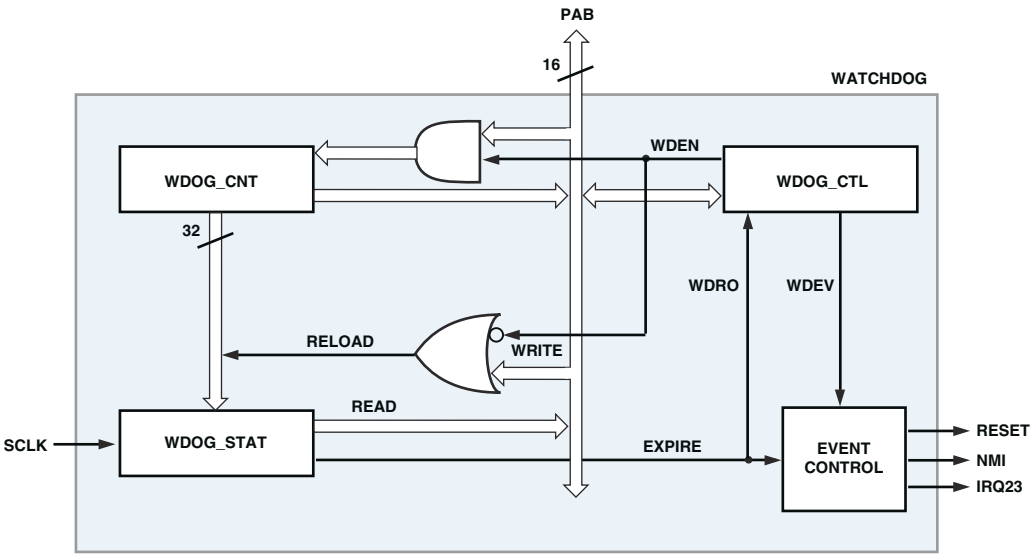


Figure 12-1. Watchdog Timer Block Diagram

External Interface

The watchdog timer does not directly interact with any pins of the chip.

Internal Interface

The watchdog timer is clocked by the system clock `SCLK`. Its registers are accessed through the 16-bit peripheral access bus PAB. The 32-bit registers `WDOG_CNT` and `WDOG_STAT` must always be accessed by 32-bit read/write operations. Hardware ensures that those accesses are atomic.

When the counter expires, one of three event requests can be generated. Either a reset or an NMI request is issued to the core event controller (CEC) or a general-purpose interrupt request is passed to the system interrupt controller (SIC).

Description of Operation

If enabled, the 32-bit watchdog timer counts downward every SCLK cycle. If it becomes 0, one of three event requests can be issued to either the CEC or the SIC. Depending on how the WDEV bit field in the WDOG_CTL register is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt.

The counter value can be read through the 32-bit WDOG_STAT register. The WDOG_STAT register cannot, however, be written directly. Rather, software writes the watchdog period value into the 32-bit WDOG_CNT register *before* the watchdog is enabled. Once the watchdog is started, the period value cannot be altered.

To start the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (WDOG_CNT). Since the watchdog timer is not yet enabled, the write to the WDOG_CNT registers automatically preloads the WDOG_STAT register as well.
2. In the watchdog control register (WDOG_CTL), select the event to generate upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register.

If software does not serve the watchdog in time, WDOG_STAT register continues decrementing until it reaches 0. Then, the programmed event is generated. The counter stops decrementing and remains at zero. Additionally, the WDRO latch bit in the WDOG_CTL register is set and can be interrogated by software in case event generation is not enabled.

When the watchdog is programmed to generate a reset, it resets the processor core and peripherals. If the NOBOOT bit in the SYSCR register was set by the time the watchdog reset the part, the chip is not rebooted. This is recommended behavior in slave boot configurations. The reset handler may evaluate the RESET_WDOG bit in the software reset register SWRST to detect a reset caused by the watchdog. For details, see Chapter 17, "System Reset and Booting".

To prevent the watchdog from expiring, software serves the watchdog by performing dummy writes to the WDOG_STAT register address in time. The values written are ignored, but the write commands cause the WDOG_STAT register to reload from the WDOG_CNT register.

If the watchdog is enabled with a zero value loaded to the counter and the WDRO bit was cleared, the WDRO bit of the watchdog control register is set immediately and the counter remains at zero without further decrements. If, however, the WDRO bit was set by the time the watchdog is enabled, the counter decrements to 0xFFFF FFFF and continues operation.

Software can disable the watchdog timer only by writing a 0xAD value (WDDIS) to the WDEN field in the WDOG_CTL register.

Watchdog Timer Registers

The watchdog timer is controlled by three registers.

- "Watchdog Count (WDOG_CNT) Register" on page 12-6
- "Watchdog Status (WDOG_STAT) Register" on page 12-7
- "Watchdog Control (WDOG_CTL) Register" on page 12-8

Watchdog Count (WDOG_CNT) Register

The watchdog count register (WDOG_CNT, shown in Figure 12-2) holds the 32-bit unsigned count value. The WDOG_CNT register must always be accessed with 32-bit read/writes.

The watchdog count register holds the programmable count value. A valid write to the watchdog count register also preloads the watchdog counter. For added safety, the watchdog count register can be updated only when the watchdog timer is disabled. A write to the watchdog count register while the timer is enabled does not modify the contents of this register.

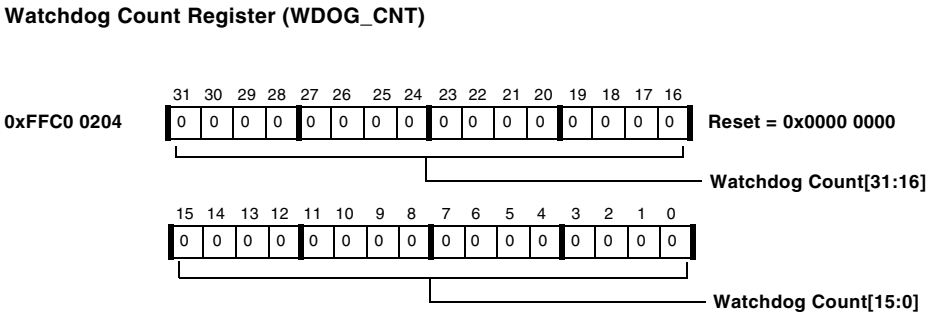


Figure 12-2. Watchdog Count Register

Watchdog Status (WDOG_STAT) Register

The 32-bit watchdog status register (WDOG_STAT, shown in Figure 12-3) contains the current count value of the watchdog timer. Reads to WDOG_STAT register return the current count value. Values cannot be stored directly in WDOG_STAT register, but are instead copied from WDOG_CNT register. This can happen in two ways:

- While the watchdog timer is disabled, writing the WDOG_CNT register preloads the WDOG_STAT register.
- While the watchdog timer is enabled, but not yet rolled over, writes to the WDOG_STAT register load it with the value in WDOG_CNT register.

Enabling the watchdog timer does not automatically reload WDOG_STAT register from WDOG_CNT register.

The WDOG_STAT register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

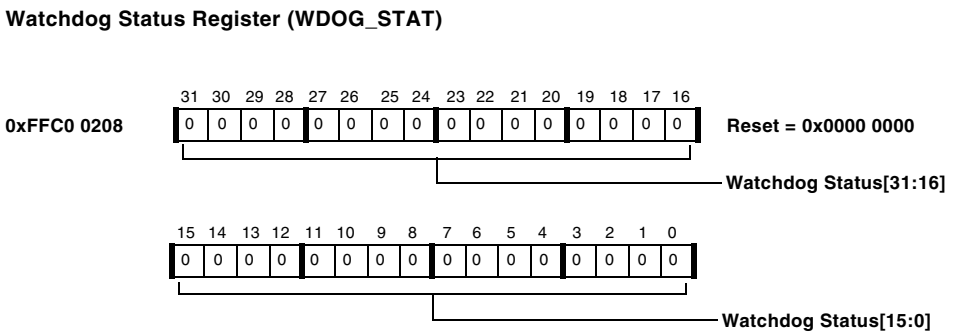


Figure 12-3. Watchdog Status Register

Watchdog Control (WDOG_CTL) Register

The watchdog control register (WDOG_CTL, shown in Figure 12-4) is a 16-bit system memory-mapped register used to control the watchdog timer.

The watchdog event (WDEV[1:0]) bit field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the system interrupt mask register (SIC_IMASK) should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The watchdog enable (WDEN[7:0]) bit field is used to enable and disable the watchdog timer. Writing any value other than the disable value (0xAD) into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the watchdog has expired by interrogating the watchdog rolled over (WDRO) status bit of the watchdog control register. This is a sticky bit that is set whenever the watchdog timer count reaches 0. It can be cleared only by writing a 1 to the bit when the watchdog has been disabled first.

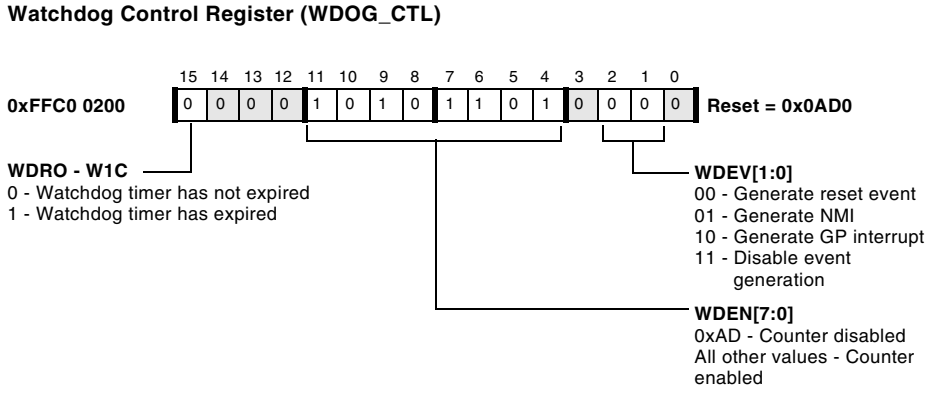


Figure 12-4. Watchdog Control Register

Programming Examples

Listing 12-1 shows how to configure the watchdog timer so that it resets the chip when it expires. At startup, the code evaluates whether the recent reset event is caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 12-1. Watchdog Timer Configuration

```
#include <blackfin.h>
#define WDOGPERIOD 0x00200000

.section L1_code;
.global _reset;
_reset:
    ...
/* optionally, test whether reset was caused by watchdog */
    p0.h=hi(SWRST);
    p0.l=lo(SWRST);
    r6 = w[p0] (z);
```

```
    CC = bittst(r6, bitpos(RESET_WDOG));
    if !CC jump _reset.no_watchdog_reset;

/* optionally, warn at system level or host device here */

_reset.no_watchdog_reset:
/* optionally, set NOBOOT bit to avoid reboot in case */
    p0.h=hi(SYSCR);
    p0.l=lo(SYSCR);
    r0 = w[p0](z);
    bitset(r0,bitpos(NOBOOT));
    w[p0] = r0;


/* start watchdog timer, reset if expires */
    p0.h = hi(WDOG_CNT);
    p0.l = lo(WDOG_CNT);
    r0.h = hi(WDOGPERIOD);
    r0.l = lo(WDOGPERIOD);
    [p0] = r0;
    p0.l = lo(WDOG_CTL);
    r0.l = WDEN | WDEV_RESET;
    w[p0] = r0;
    ...
    jump _main;
_reset.end:
```

The subroutine shown in Listing 12-2 can be called by software to service the watchdog. Note that the value written to the `WDOG_STAT` register does not matter.

Listing 12-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Listing 12-3 is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 12-3. Watchdog Restarted by Interrupt Service Routine

```
isr_watchdog:
    [--sp] = astat;
    [--sp] = (p5:5, r7:7);
    p5.h = hi(WDOG_CTL);
    p5.l = lo(WDOG_CTL);
    r7.l = WDDIS;
    w[p5] = r7;
    bitset(r7, bitpos(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:
```

13 ROTARY COUNTER

This chapter describes the rotary (up/down) counter, which provides support for manually-controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial encoders.

This chapter describes the rotary counter and includes the following sections:

- "Overview" on page 13-1
- "Interface Overview" on page 13-3
- "Description of Operation" on page 13-4
- "Functional Description" on page 13-8
- "Programming Mode" on page 13-24
- "Rotary Counter Registers" on page 13-24
- "Programming Examples" on page 13-33

Overview

The primary purpose of the rotary counter is to convert pulses from incremental position encoders into data that is representative of the actual position. This is done by integrating (counting) pulses on one or two inputs.

Since integration provides relative position, some devices also feature a zero position input (zero marker) that can be used to establish a reference point or alternative to verify that the acquired position does not drift over time.

In addition, the incremental position information can be used to determine speed, if the time intervals are measured.

The rotary counter interface provides various and flexible ways to establish position information. When used in conjunction with the general-purpose (GP) timer block, the rotary counter interface allows for the acquisition of coherent position/timestamp information that enables speed calculation.

Features

The rotary counter includes the following features:

- 32-bit rotary counter
- Quadrature encoder mode (gray code)
- Binary encoder mode
- Alternative frequency-direction mode
- Timed direction and up/down counting modes
- Zero marker/pushbutton support
- Capture event timing in association with GP timer
- Boundary comparison and boundary setting features
- Input pin noise filtering (debouncing)
- Flexible error detection/signaling

Interface Overview

A block diagram of the rotary counter interface is shown in Figure 13-1. There are two input pins, the count up and direction (CUD) pin and the count down and gate (CDG) pin, that accept various forms of incremental inputs and are processed by the 32-bit counter. The third input, count zero marker (CZM), is the zero marker input. The module interfaces to the processor by way of the peripheral access bus (PAB) and can optionally generate an interrupt request through the IRQ line. There is also an output that can be used by the timer module to generate timestamps on certain events.

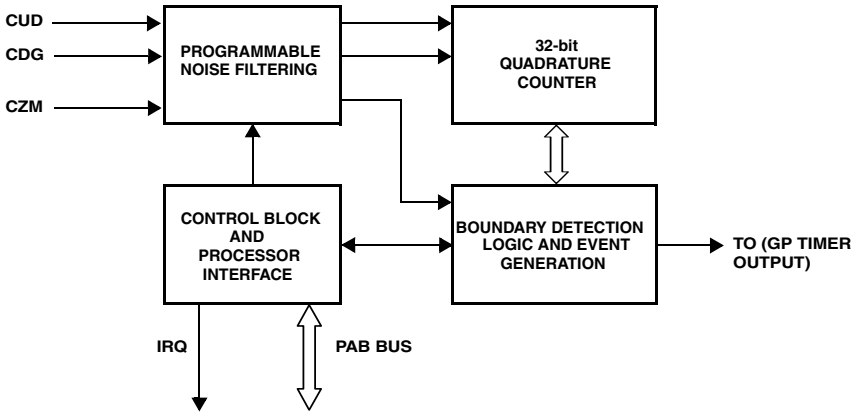


Figure 13-1. Block Diagram of the Rotary Counter Interface

The timer output signal is connected internally to the alternate capture input (TACI6) of the general-purpose timer 6. The interrupt signal goes to the IRQ68 input of the SIC2 controller.

Description of Operation

The rotary encoder block has five modes of operation that are described in this section.

With the exception of the timed direction mode, the rotary timer block can operate in conjunction with the GP timer block in order to capture additional timing information (timestamps) associated with events detected by this block.

The third input (CZM) may be used as a zero marker or to sense the pressing of a pushbutton. Refer to "Zero Marker (Pushbutton) Operation" on page 13-12 for more details.

The three input pins may be filtered (debounced) prior to being evaluated by the rotary encoder. Refer to "Input Noise Filtering (Debouncing)" on page 13-8 for more details.

The encoder block also features a flexible boundary comparison. In all of the operating modes, the counter can be compared to an upper and lower limit. A variety of actions can be taken when these limits are reached. Refer to "Boundary Comparison Modes" on page 13-13 for more details.

Quadrature Encoder Mode

In this mode, the CUD:CDG inputs expect a quadrature-encoded signal that is interpreted as a 2-bit gray code. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The register CNT_COUNTER contains the number of transitions that have occurred. Refer to Figure 13-2 for more details.

Figure 13-2 shows an example of a series of count up events which is causing CNT_COUNTER to increment.

Optionally, an interrupt is generated if both inputs change within one SCLK cycle. Such transitions are not allowed by gray coding. Therefore, the register CNT_COUNTER remains unchanged and an error condition is signaled.

CNT_COUNTER register value	-4	-3	-2	-1	0	+1	+2	+3	+4
CDG:CUD inputs	00	01	11	10	00	01	11	10	00

Figure 13-2. Quadrature Events and Counting Mechanism

It is possible to reverse the count direction of the gray-coded signal. This can be achieved by enabling the polarity inverter of either the CUD pin or the CDG pin, inverting both pins does not alter the behavior. This feature can be enabled with the CDGINV and CUDINV bits in the CNT_CONFIG register.

As an example, if the CDG:CUD inputs are 00 respectively and the next transition is to 01, this would normally increment the counter as is seen in Figure 13-2. If the CUD polarity is inverted this generates a received input of 01 followed by 00. This will result in a decrement of the counter, altering the behavior of the connected hardware.

Binary Encoder Mode

This mode is almost identical to the previous mode, with the exception that the CUD:CDG inputs expect a binary-encoded signal. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The register CNT_COUNTER contains the number of transitions that have occurred. Refer to Figure 13-3.

In Figure 13-3, a series of binary up count events are causing the CNT_COUNTER register to increment.

Optionally, an interrupt is generated if the detected code steps by more than 1 (in binary arithmetic) within one SCLK cycle. Such transitions are considered erroneous. Therefore, the register CNT_COUNTER remains unchanged and an error condition is signaled.

Reversing the CUD and CDG pin polarity has a different effect for the binary encoder mode than from the quadrature encoder mode. Inverting the polarity of the CUD pin only or inverting both the CUD and CDG pins result in reversing the count direction.

CNT_COUNTER register value	-4	-3	-2	-1	0	+1	+2	+3	+4
CDG:CUD inputs	00	01	10	11	00	01	10	11	00

Figure 13-3. Binary Events and Counting Mechanism

Rotary Counter Mode

In this general-purpose mode, the counter is incremented or decremented at every active edge of the input pins.

If an active edge is detected at the CUD input, the counter increments. The active edge can be selected by way of the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge increments the counter. If the configuration bit is set, a falling edge increments the counter.

If an active edge is detected at the CDG input, the counter decrements. The active edge can be selected by way of the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge decrements the counter. If the configuration bit is set, a falling edge decrements the counter.

If simultaneous edges occur on pin CDG and pin CUD, the counter remains unchanged and both up-count and down-count events are signaled in the CNT_STATUS register.

Direction Counter Mode

In this mode the CUD input pin is used to determine direction and the CDG input is used as a gate.

In this general-purpose mode, the counter is incremented or decremented at every active edge of the CDG input pin.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by way of the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input selects the direction to increment, a low input selects the direction to decrement. If the configuration bit is set, the polarity is inverted.

If an active edge is detected at the CDG input, the counter value changes by one in the selected direction. The active edge can be selected by way of the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge decrements the counter. If the configuration bit is set, a falling edge decrements the counter.

Timed Direction Mode

In this general-purpose mode, the counter is incremented or decremented at each SCLK cycle.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by way of the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment the counter, a low input decrements it. If the configuration bit is set, the polarity is inverted.

The CDG pin can be used to gate the clock. The polarity can be selected by way of the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a high CDG input enables the counter, a low input will stop it. If the configuration bit is set, the polarity is inverted.

Functional Description

The following sections describe the rotary counter in more detail.

Input Noise Filtering (Debouncing)

The rotary inputs are asynchronous to the system clock so hardware synchronizes them internally before using. This synchronization causes a fixed delay of a few clocks before any actions result from the toggling of the inputs.

Because of the synchronization, the minimum pulse width of the input signals must be the period of the system clock. For signals which don't require debouncing, the maximum input frequency is the same as the system clock.

In all modes, the three input pins can be optionally filtered in order to present clean signals to the subsequent rotary encoder logic. This feature can be enabled or disabled by way of the DEBE bit in the CNT_CONFIG register.

The filtering mechanism is implemented using counters for each pin. The counter for each pin is initialized from the DPRESCALE field of the CNT_DEBOUNCE register. Whenever a transition is detected on a pin, the corresponding counter starts counting up to the programmed number of SCLK cycles. The state of the pin is then latched after time t_{FILTER}, as determined by the equation below and passed on to the subsequent logic. The 5-bit DPRESCALE field in the CNT_DEBOUNCE register (see Figure 13-12 on page 13-31) is used to program the desired cycle number and therefore the debouncing time. The number of SCLK cycles used to program the counters for each pin can be selected in eighteen steps by way of this register, see Table 13-1 on page 13-10.

The time t_{filter} is determined, given SCLK and the DPRESCALE value contained in the CNT_DEBOUNCE register, by the following formula:

$$t_{FILTER} = 128 \times (2^{DPRESCALE} \div SCLK)$$

where, DPRESCALE can contain values from 0 (minimum filtering) to 17 (maximum filtering).

Figure 13-4 shows the filtering operation for the CUD pin.

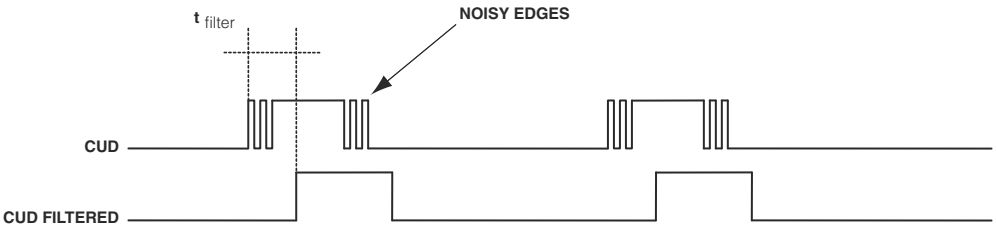


Figure 13-4. Programmable Noise Filtering

Assuming an SCLK frequency of 133 MHz, the filter time range is shown by the following two equations, Figure 13-5 on page 13-10, Table 13-1 on page 13-10, and Table 13-2 on page 13-11:

DPRESCALE = 0b00000

$$t_{FILTER} = 128 \times 1 \times 7.5ns = 960ns = 1\mu s(approx)$$

DPRESCALE = 0b10001

$$t_{FILTER} = 128 \times 131072 \times 7.5ns = 125829\mu s = 126ms(approx)$$

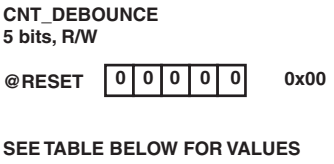


Figure 13-5. Filtering Range

Table 13-1. DPRESCALE Filtering Range

DPRESCALE	Cycles	Debounce Time (approximately)
00000	1x	1 µs
00001	2x	2 µs
00010	4x	4 µs
00011	8x	8 µs
00100	16x	16 µs
001001	32x	32 µs
00110	64x	64 µs
00111	128x	128 µs
01000	256x	256 µs
01001	512x	512 µs
01010	1024x	1 ms
01011	2048x	2 ms
01100	4096x	4 ms
01101	8192x	82 ms
01110	16384x	16 ms
01111	32768x	32 ms
10000	65536x	64 ms
10001	131072x	126 ms

Table 13-2. Prescale Value Programming to Debounce Filter Circuit

Bit Location	Name	Type	Function	
4:0	DPRES-CALE	R/W	These bits are used to program the prescale value to the debounce filter circuit in the counter module. The predefined count value for "x" (128) determines the number of SCLK cycles to be counted.	
			b#00000	1x cycles = 128 SCLK cycles
			b#00001	2x cycles = 256 SCLK cycles
			b#00010	4x cycles = 512 SCLK cycles
			b#00011	8x cycles = 1024 SCLK cycles
			b#00100	16x cycles = 2048 SCLK cycles
			b#00101	32x cycles = 4056 SCLK cycles
			b#00110	64x cycles = 8112 SCLK cycles
			b#00111	128x cycles = 16224 SCLK cycles
			b#01000	256x cycles = 32448 SCLK cycles
			b#01001	512x cycles = 64896 SCLK cycles
			b#01010	1024x cycles = 129792 SCLK cycles
			b#01011	2048x cycles = 259584 SCLK cycles
			b#01100	4096x cycles = 519168 SCLK cycles
			b#01101	8192x cycles = 1038336 SCLK cycles
			b#01110	16384x cycles = 2076672 SCLK cycles
			b#01111	32768x cycles = 4153344 SCLK cycles
			b#10000	65536x cycles = 8306688 SCLK cycles
			b#10001	131072x cycles = 16613376 SCLK cycles
			b#10010	10010b - 11111b: Reserved

Zero Marker (Pushbutton) Operation

The CZM input pin can be used to sense the zero marker output of a rotary device or detect pressing of a pushbutton. There are four programming schemes all of which are functional in all counter modes. They are listed as follows:

- **Pushbutton mode** This mode is enabled by setting the CZMIE bit in the CNT_IMASK register. An active edge at the CZM input sets the CZMII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this generates an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register: rising edge if cleared, falling edge if set to one.
- **Zero-marker-zeros-counter mode** This mode is enabled by setting the ZMZC bit in the CNT_ CONFIG register. An active level at the CZM input clears the CNT_COUNTER register and holds it until the CZM pin is deactivated. In addition, if enabled by the CZMZIE bit in the CNT_IMASK register, this mode sets the CZMZII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this generates an interrupt request. The active level is selected by the CZMINV bit in the CNT_CONFIG register: active high if cleared, active low if set to one.
- **Zero-marker-error mode** This mode is used to detect discrepancies between the counter value and the zero marker output of certain rotary encoder devices. It is enabled by setting the CZMEIE bit in the CNT_IMASK register. When an active edge is detected at the CZM input pin, the four LSBs of the CNT_COUNTER register are compared to zero. If they are not zero, a mismatch is signaled by way of the CZMEII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this mode generates an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register: rising edge if cleared, falling edge if set to one.

- **Zero-once mode** This mode is used to perform an initial reset of the counter value when an active zero marker is detected. After that, the zero marker is ignored (the counter is not reset anymore). This mode is enabled by setting the W1ZMONCE bit in the CNT_COMMAND register. The CNT_COUNTER register and the W1ZMONCE bit are cleared on the next active edge on the CZM pin. Thus, the W1ZMONCE bit can be read to check whether the event has already occurred, if desired. The active edge of the CZM pin is selected by the CZMINV bit in the CNT_CONFIG register: rising edge if cleared, falling edge if set to one.

Boundary Comparison Modes

The rotary encoder block includes two boundary registers, CNT_MIN (lower) and CNT_MAX (upper). The counter value is compared to the lower and upper boundary. Depending on which mode is selected, different actions are taken if the count value reaches either of the boundary values.

For all boundary modes, compares do not occur if the change to CNT_MIN/CNT_MAX/CNT_COUNTER was due to a software event. Software events include writing these registers, or events caused by writing the CNT_COMMAND register. Boundary compare events ONLY occur due to up/down actions from the counter. This includes setting MINCII/MAXCII and zeroing the counter on a compare to either CNT_MIN or CNT_MAX.

There are four boundary modes:

- **Boundary-compare mode** The two boundary registers are simply compared to the CNT_COUNTER register. If CNT_COUNTER after incrementing equals CNT_MAX, the MAXCII bit in the CNT_STATUS register is set. If the MAXCIE bit in the CNT_IMASK register is set, an interrupt request is generated. Similarly CNT_COUNTER after decrementing equals CNT_MIN, the MINCII status bit is set. If the MINCIE bit in the CNT_IMASK register is set, an interrupt request is generated. The

MAXCII and MINCII bits are not set if the CNT_MAX and CNT_MIN registers are updated by software. For MINCII and MAXCII to be set, all that needs to happen is for CNT_COUNTER to equal them, regardless of the direction. As an example, if CNT_MIN=2 and CNT_COUNTER=1 and an up event occurs, MINCII will still be set. Likewise, if CNT_MAX=2, CNT_COUNTER=3 and a down event occurs, MAXCII will still be set.

For the special case of CNT_MIN equals CNT_MAX, if CNT_COUNTER reaches the value in the boundary register both MINCII and MAXCII are set.

- **Boundary-zero mode** This mode is similar to the boundary-compare mode. In addition to setting the status bits and requesting interrupts, the counter value in the CNT_COUNTER register is also set to zero.
- **Boundary auto-extend mode** In this mode, the boundary registers are modified by hardware whenever the counter value reaches either of them. At startup, the application software should set both boundary registers to the initial CNT_COUNTER value. The CNT_MAX register is loaded with the current CNT_COUNTER value if the latter increments beyond the CNT_MAX value. Similarly, the CNT_MIN register is loaded with the CNT_COUNTER value if the latter decrements below the CNT_MIN value. This mode may be used to keep track (in hardware) of the widest angle the wheel ever reported, even if the software did not serve interrupts. The MAXCII and MINCII status bits are still set when the counter value matches the boundary register, not only when it extends the boundary.

  In this mode it is envisioned that software would never change CNT_MIN or CNT_MAX by writing to them or an action from the CNT_COMMAND register. If software does this, the behavior is best described by a few examples:

Example 1: CNT_MAX=2, CNT_COUNTER=1. With three up events, CNT_COUNTER=CNT_MAX=4. Now if software writes CNT_MAX=2, the rotary will not auto extend until CNT_COUNTER decrements back down to two, then increments again.

Example 2: CNT_MIN=2, CNT_COUNTER=3. With three down events CNT_COUNTER=CNT_MIN=0. Now if software writes CNT_MIN=2, the rotary will not auto extend until CNT_COUNTER increments back up to two, then decrements again.

- **Boundary-capture mode** In this mode, the CNT_COUNTER value is latched into the CNT_MIN register at one detected edge of the CZM input pin, and latched into CNT_MAX at the opposite edge. If the CZMINV bit in the CNT_CONFIG register is cleared, a rising edge captures into CNT_MIN and a falling edge into CNT_MAX. If the CZMINV bit is set, the edges are inverted. The MAXCII and MINCII status bits report the capture event.

  The comparison is performed with signed arithmetic. The boundary registers and the counter value are all treated as signed integer values.

Rotary Encoder Events: Control and Signaling

There are a total of 11 events that can be signaled to the processor by way of status information and optional interrupt requests. The interrupts are enabled by the respective bits in the CNT_IMASK register. Dedicated status bits in the CNT_STATUS register report events. When an interrupt from the rotary encoder is acknowledged, the application software is responsible for correct interpretation of the events. It is recommended to logically AND the content of the CNT_IMASK and CNT_STATUS registers to identify pending interrupts. Interrupt requests are cleared by write-one-to-clear (W1C) operations to the CNT_STATUS register. Hardware does not clear the status

bits automatically, unless the counter module is disabled. There are four boundary modes. Status bits are available in any of the counter modes discussed in "Description of Operation" on page 13-4.

Illegal Gray/Binary Code Events (Two-Step Detection)

As described in the quadrature encoder mode and binary encoder mode sections, illegal transitions can be detected in these two modes. If this event occurs, the ICII status bit is set. If enabled by the ICIE bit, an interrupt request is generated. The ICIE bit should only be used (set) in these two modes.

Up/Down Count Events

The UCII status bit informs whether the counter is incremented. Similarly, the DCII bit reports decrements. The two events are independent. For instance, if the counter first increments by one and then decrements by two, both bits remain set, even though the resulting counter value shows a decrement by one. In rotary counter mode, hardware may detect simultaneous active edges on the CUD and CDG inputs. In that case, the CNT_COUNTER remains unchanged, but both the UCII and DCII bits are set.

Interrupt requests for these events may be enabled through the UCIE and DCIE bits. This feature should be used carefully when the counter is clocked at high rates. This is especially critical when the counter operates in DIR_TMR mode, as interrupts would be generated every SCLK cycle.

These events can also be used for additional push buttons, if rotary encoder features are not needed. When rotary counter mode is enabled, these count events can be used to report interrupts from push buttons that connect to the CUD and CDG inputs.

Zero Count Events

The CZEROII status bit indicates that the CNT_COUNTER has reached a value equal to 0x0000 0000 after an increment or decrement. This bit is not set when the counter value is set to zero directly by way of a software write (write to CNT_COUNTER or setting the W1LCNT_ZERO bit in the CNT_COMMAND register). If enabled by the CZEROIE bit, an interrupt request is generated.

Overflow Events

There are two status bits that indicate whether the signed counter register has overflowed from a positive to a negative value or vice versa.

The COV31II bit reports that the 32-bit CNT_COUNT register has either incremented from 0x7FFF FFFF to 0x8000.0000 or decremented from 0x8000.0000 to 0x7FFF FFFF. If enabled by the COV31IE bit, an interrupt request is generated.

Similarly, in applications where only the lower 16 bits of the counter are of interest, the COV15II status bit reports counter transitions from 0xxxxx 7FFF to 0xxxxx 8000 or reversed. If enabled by the COV15IE bit, an interrupt request is generated.

Boundary Match Events

The MINCII and MAXCII status bits report boundary events as described in "Boundary Comparison Modes" on page 13-13. These bits are not set if the CNT_COUNTER, CNT_MAX or CNT_MIN registers are updated by software or the CNT_COMMAND register is written to.

The MINCIE and MAXCIE bits in the CNT_IMASK register enable interrupt generation on boundary events.

Zero Marker Events

There are three status bits CZMII, CZMEII and CZMZII associated with zero marker events, as described in "Zero Marker (Pushbutton) Operation" on page 13-12. Each of these events can optionally generate an interrupt request, if enabled by the corresponding CZMIE, CZMEIE and CZMZIE bits in the CNT_IMASK register.

Capturing Timing Information (Using the General-Purpose Timer)

To calculate speed, many applications may wish to measure the time between two count events—in addition to accurately counting encoder pulses. For more accuracy, particularly at very low speeds, it is also necessary to obtain the time that has elapsed since the last count event. This additional information allows for estimating how much the GP counter has advanced since the last counter event.

For this purpose, the GP counter has an internal signal that connects to the alternate capture input (TACIx) of one of the GP timers. It is functional in all modes, with the exception of the timed direction mode. Refer to "Internal Interfaces" in Chapter 9, "General-Purpose Ports" for information regarding which GP timer(s) are associated with which GP counter module(s) for your device.

In order to use the timing measurements, the associated GP timer must be used in the WDTH_CAP mode. The alternate capture input is selected by setting the TIN_SEL bit in the GP timer's TIMER_CONFIG register. For more information see Chapter 10, "General-Purpose Timers".

For this purpose, the rotary counter has an internal timer output that connects to the alternate capture inputs (TACIx) of one of the timers as explained in "Interface Overview" on page 13-3. It is functional in all modes, with the exception of the timed direction mode.

In order to use the timing measurements, the associated timer must be used in pulse width count and capture mode (WDTH_CAP). The alternative capture input is selected by setting the TIN_SEL bit in the timer's configuration register. For more information about the GP Timers and their operating modes refer "Capturing Timings from the GP Counter Module" on page 10-34.

Capturing Time Interval Between Successive Counter Events

When the only timing information of interest is the interval between successive count events, the associated timer should be programmed in WDTH_CAP mode with PULSE_HI = 1, PERIOD_CNT = 1 and TIN_SEL = 1. Typically, this information is sufficient if the speed of rotary encoder events is known not to reach very low values. Figure 13-6 on page 13-20 shows the operation of the rotary encoder module and the GP timer in this mode. TO generates a pulse every time a count event occurs. The general-purpose timer will update the TIMERx_PERIOD register with the period (measured from rising edge to rising edge) of the TO signal. The TIMERx_PERIOD register is updated at every rising edge of the TO signal and contains the number of system clock (SCLK) cycles that have elapsed since the previous rising edge.

Incidentally, the TIMERx_WIDTH register is also updated at the same time, but is generally of no interest in this mode of operation. If no reads of the CNT_COUNTER register occur between counter events, the TIMERx_WIDTH register only contains the width of the TO pulse. If a read of the CNT_COUNTER has occurred between events, the TIMERx_WIDTH register will contain the time between the read of the CNT_COUNTER and the next event.

This mode can also be used with PULSE_HI = 0. In this case, the period of TO is measured between falling edges. It will result in the same values as in the previous case, only the latching occurs one SCLK cycle later.

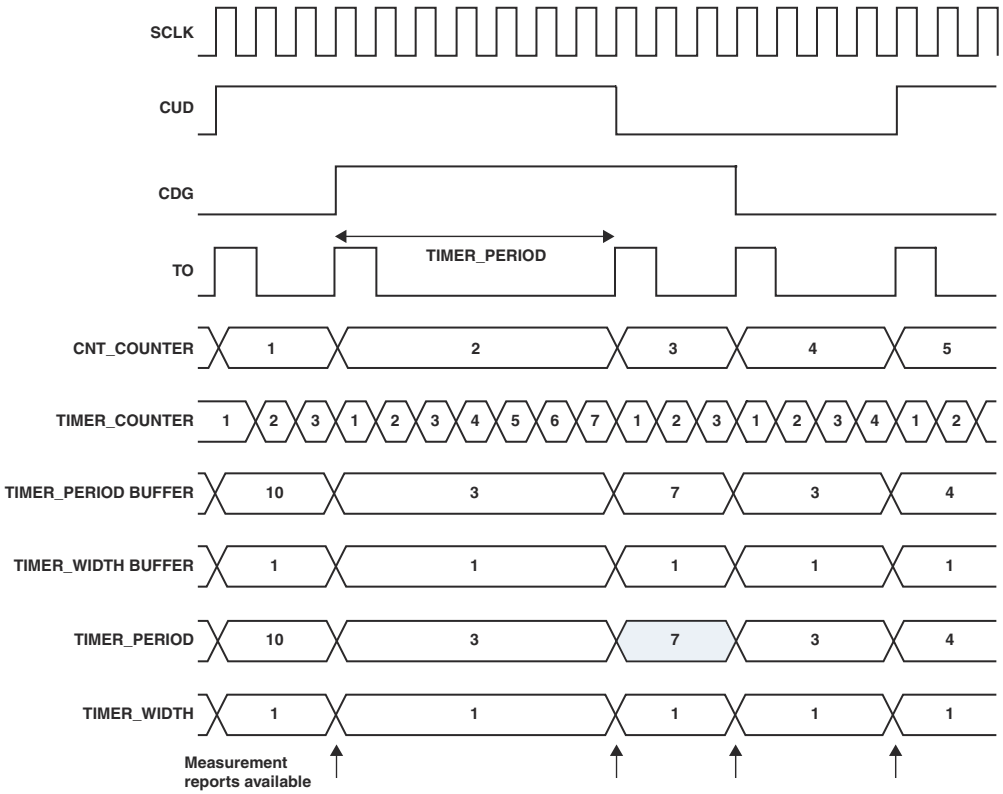


Figure 13-6. Capturing Counter Event Time Intervals

Capturing Counter Interval and CNT_COUNTER Read Timing

It is possible to also capture the time elapsed since the last count event. In this mode, the associated timer should be programmed in `WDTH_CAP` mode with `PULSE_HI` = 0, `PERIOD_CNT` = 0 and `TIN_SEL` = 1. Typically, this additional information is used to estimate the advancement of the rotary encoder since the last count event, if the speed is very low. Figure 13-7

shows the operation of the rotary encoder module and the general-purpose timer in this mode. TO generates a pulse every time a count event occurs. In addition, when the processor reads the CNT_COUNTER register, the TO signal presents a pulse which is extended (high) until the next count event. The general-purpose timer will update the TMRx_PERIOD register with the period (measured from falling edge to falling edge, because PULSE_HI = 0) of the TO signal. The TMRx_WIDTH register is updated with the pulse width (the portion where TO is low, again because PULSE_HI = 0). Both registers are updated at every rising edge of the TO signal (because PERIOD_CNT = 0). Therefore, the period register contains the period between the last two count events and the width register contains the time since the last count event and the read of the CNT_COUNTER register, both measured in number of system clock (SCLK) cycles.

The result is that when reading the CNT_COUNTER register, the two time measurements are also latched and the user has a coherent triplet of information to calculate speed and position.

Restrictions apply to the use of the TO signal in terms of speed. Therefore, the user must take care to not operate at very high count events. For instance, if CNT_COUNTER is incremented/decremented every SCLK cycle (timed direction mode), the TO signal is incorrect.

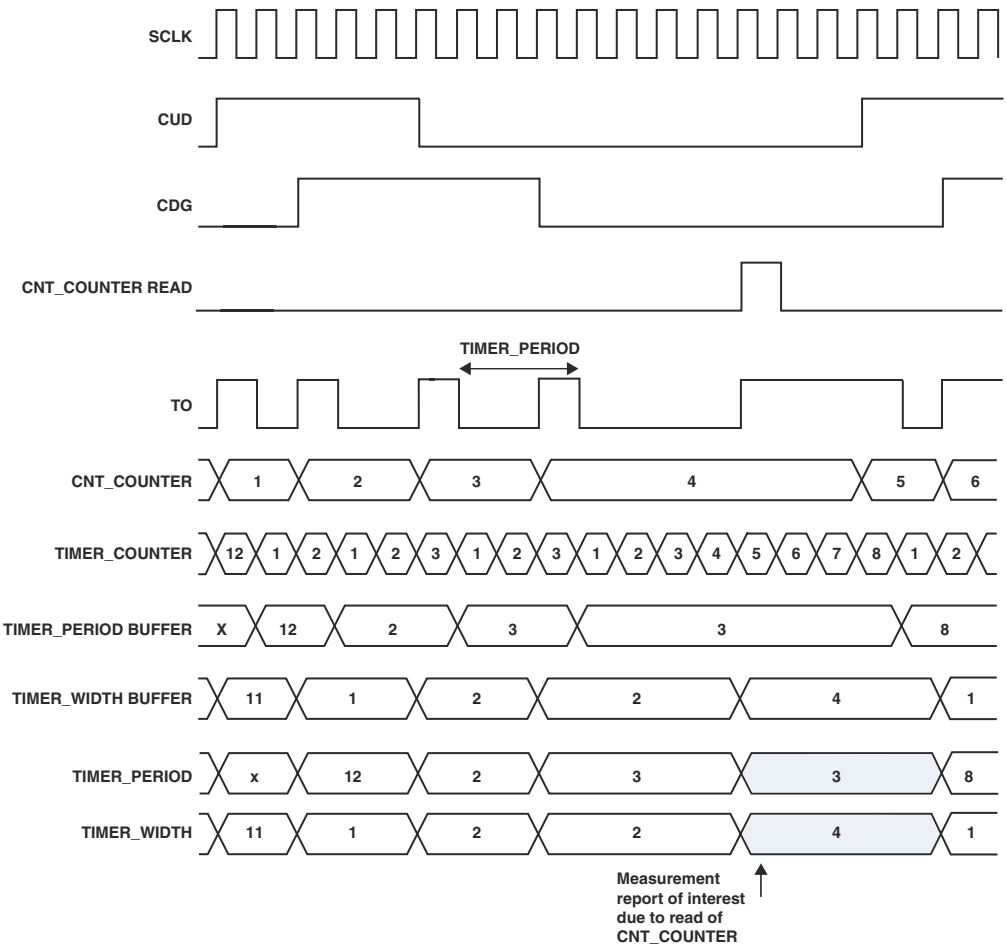


Figure 13-7. Capturing Counter Event Time Intervals and Last Event Time Interval

Counter Commands

In order to facilitate initialization of the peripheral, a register is provided to perform various operations such as zeroing a counter register, copying or swapping boundary registers and so on. These actions are taken by writing a 1 to the appropriate bit in the CNT_COMMAND register.

The CNT_COUNTER, CNT_MIN and CNT_MAX registers can be initialized to zero by writing a 1 to the W1LCNT_ZERO, W1LMIN_ZERO and W1LMAX_ZERO fields. In addition to clearing registers, the boundary registers can be modified in a number of ways. The current counter value in CNT_COUNT can be captured and loaded into either of the two boundary registers CNT_MAX and CNT_MIN to create new boundary limits. This is performed by setting the W1LMAX_CNT and W1LMIN_CNT bits. Alternatively the counter can be loaded from CNT_MAX or CNT_MIN through the W1LCNT_MAX and W1LCNT_MIN bits. It is also possible to transfer the current CNT_MAX into CNT_MIN or vice versa through the W1LMIN_MAX and W1LMAX_MIN bits. The final supported operation is the ability to only have the zero marker clear the CNT_COUNT register once as described in "Zero Marker (Pushbutton) Operation" on page 13-12.

It is possible for multiple actions to be performed simultaneously by setting multiple bits in the CNT_COMMAND register. The bits associated with each command have been grouped together such that all bits that involve a write to the CNT_COUNTER register are located within the bits 3:0 of the CNT_COMMAND register. All commands that involve a write to the CNT_MIN register are located within bits 7:4 of the CNT_COMMAND register and all commands that involve a write to the CNT_MAX register are located within bits 11:8 of the CNT_COMMAND register. Refer to the register diagram (Figure 13-11 on page 13-30) for more details.

A maximum of three commands can be issued at any one time, excluding the W1ZMONCE command. No two commands issued simultaneously can involve a load to the same counter register. The following commands must be used exclusively: W1LCNT_MIN, W1LCNT_MAX, and W1LCNT_ZERO. Never set more than one of them at

the same time. The same requirement stands true for W1LMAX_MIN, W1LMAX_CNT and W1LMAX_ZERO and also for W1LMIN_MAX, W1LMIN_CNT, and W1LMIN_ZERO.

Programming Mode

In a typical application, the programmer initializes the rotary encoder to the desired mode, without enabling it. Normally the events of interest are processed by way of interrupts rather than by polling the status bit. Therefore, clear all status bits and activate the generation of interrupt requests using the CNT_IMASK register. Set up the peripheral interrupt controller and core interrupts. If timing information is required, set up the appropriate timer in the WDTH_CAP mode with the settings described in "Capturing Timing Information (Using the General-Purpose Timer)" on page 13-18. Then, enable interrupts and the peripheral itself.

Rotary Counter Registers

The rotary encoder interface has eight memory-mapped registers (MMRs) that regulate its operation.

Refer to Table 13-3 for an overview of all MMRs associated with the rotary encoder interface.

Descriptions and bit diagrams for MMRs are provided in the following sections.

Table 13-3. Counter Module Register Overview

Address	Register Name	Description	Notes
0xFFC0 4200	CNT_CONFIG	"Configuration (CNT_CONFIG) Register" on page 13-26	16 bits R/W Reset = 0x0000
0xFFC0 4204	CNT_IMASK	"Interrupt Mask (CNT_IMASK) Register" on page 13-28	16 bits R/W Reset = 0x0000
0xFFC0 4208	CNT_STATUS	"Status (CNT_STATUS) Register" on page 13-28	16 bits R/W1C Reset = 0x0000
0xFFC0 420C	CNT_COMMAND	"Command (CNT_COMMAND) Register" on page 13-29	16 bits R/W1ACTION Reset = 0x0000
0xFFC0 4210	CNT_DEBOUNCE	"Debounce Prescale (CNT_DEBOUNCE) Register" on page 13-30	16 bits R/W Reset = 0x0000
0xFFC0 4214	CNT_COUNTER	"Counter (CNT_COUNTER) Register" on page 13-31	32 bits R/W (16/32 bits) Reset = 0x0000 0000
0xFFC0 4218	CNT_MAX	"Boundary (CNT_MIN and CNT_MAX) Registers" on page 13-32	32 bits R/W (16/32 bits) Reset = 0x0000 0000
0xFFC0 421C	CNT_MIN	"Boundary (CNT_MIN and CNT_MAX) Registers" on page 13-32	32 bits R/W (16/32 bits) Reset = 0x0000 0000

Configuration (CNT_CONFIG) Register

The configuration (CNT_CONFIG) register is used to configure counter modes and input pins and to enable the peripheral. It can be accessed at any time with 16-bit read and write operations.

To avoid false glitches on startup, write all bits in CNT_CONFIG first, followed by a second write to the register which enables the counter (CNTE = 1).

Boundary Register Mode

Since CUD, CDG, and CZM input pins are muxed with other pins, these pins might be used for a function other than rotary counter. Specifically:

- If the application needs only the pushbutton (CZM) function, then write INPDIS = 0 to ignore CUD and CDG. This allows a debounced pushbutton interrupt source.
- If the application needs just the rotary pins CUD and CDG, but not the CZM, then write INPDIS = 1. Then ensure your software does not enable any of the pushbutton functions in the rotary counter registers.

For further information, see BNDMODE bit in Figure 13-8.

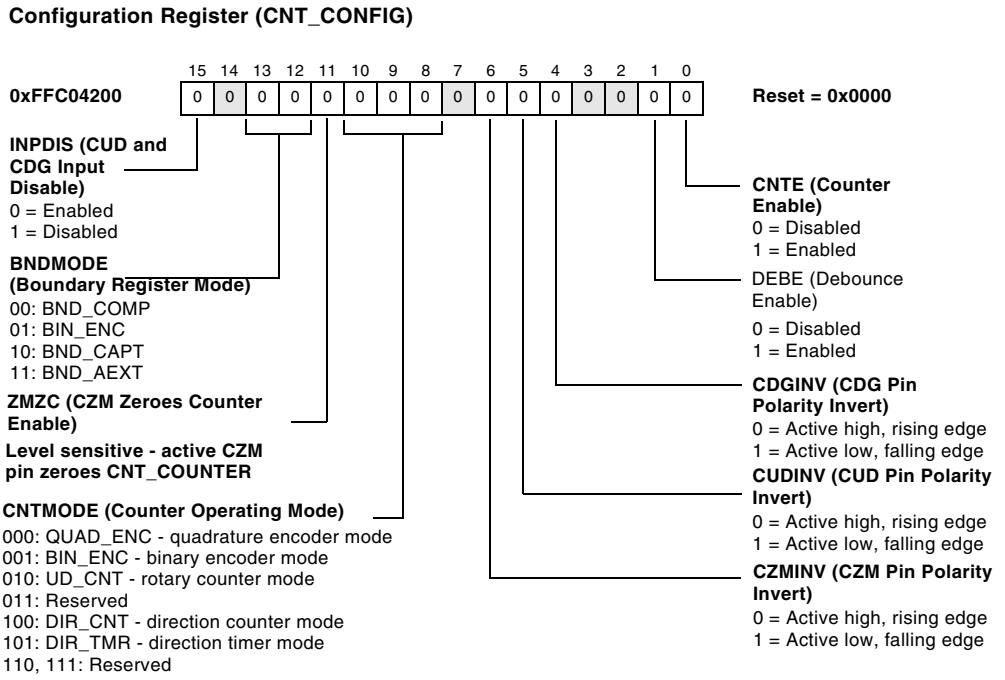


Figure 13-8. Configuration Register

Interrupt Mask (CNT_IMASK) Register

The interrupt mask (CNT_IMASK) register is used to enable interrupt request generation from each of the eleven events (See Figure 13-9). It can be accessed at any time with 16-bit read and write operations.

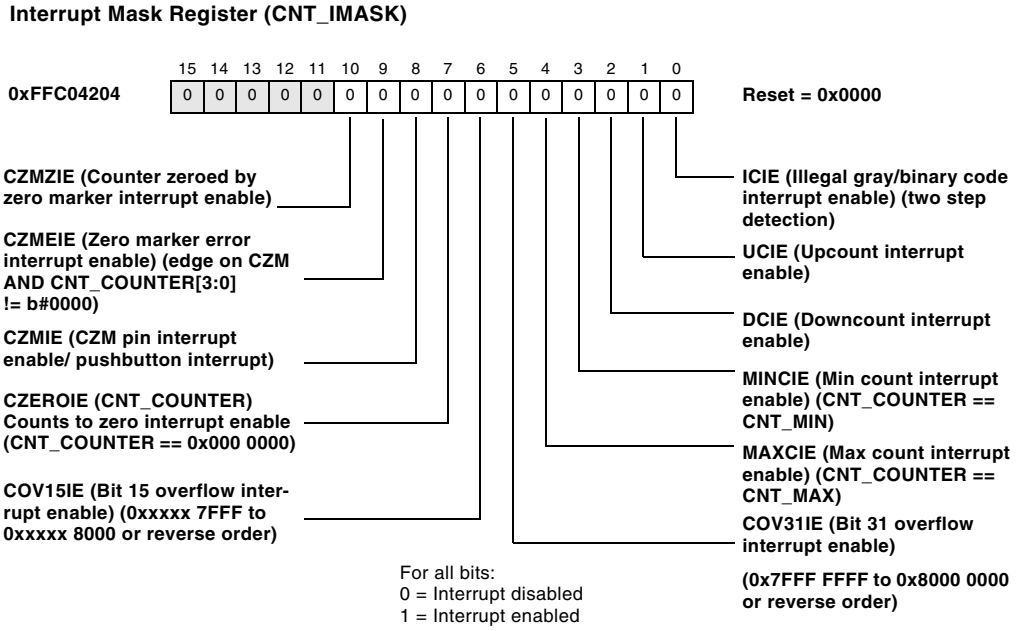


Figure 13-9. Interrupt Mask Register

Status (CNT_STATUS) Register

The status (CNT_STATUS) register provides status information for each of the eleven events where 0 = no interrupt pending and 1 = interrupt pending (See Figure 13-10). When an event is detected, the corresponding bit in this register is set. It remains set until either software writes a 1to the bit (write-1-to-clear) or the rotary encoder peripheral is disabled.

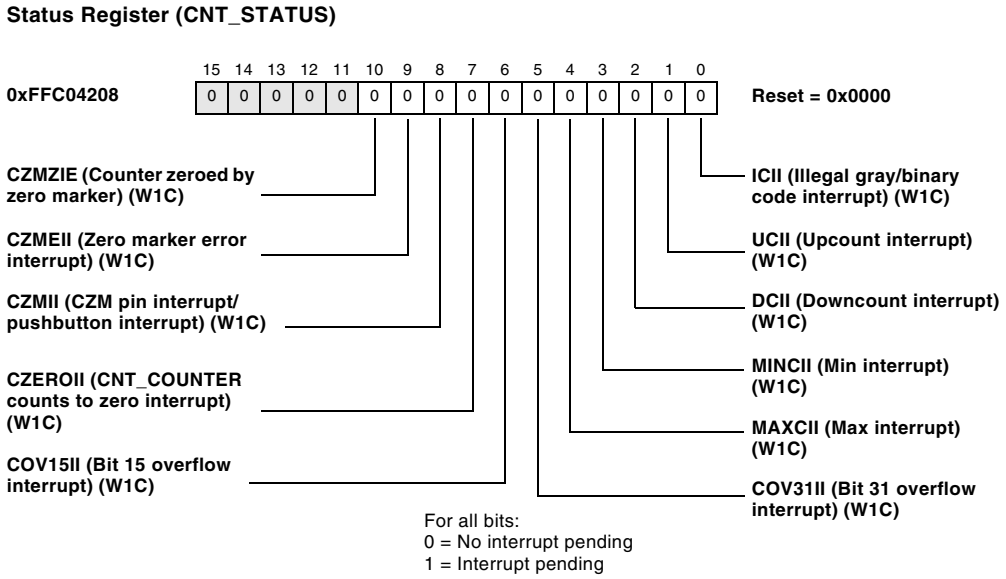


Figure 13-10. Status Register

Command (CNT_COMMAND) Register

The command (`CNT_COMMAND`) register is used to perform various actions that are needed occasionally. Each bit performs the indicated action when a 1 is written to it (See Figure 13-11).

Read operations from this register do not return meaningful values. One exception is the `W1ZONCE` bit. It is the only bit that returns a value if the register is read. A one indicates that the bit is set by software before, but no zero marker event is detected on the `CZM` pin yet. Refer to "Zero Marker (Pushbutton) Operation" on page 13-12 for more details.

Note that `W1LCNT_MIN`, `W1LCNT_MAX` and `W1LCNT_ZERO` have to be used exclusively. Never set more than one of them at the same time. The same requirement stands for `W1LMAX_MIN`, `W1LMAX_CNT` and `W1LMAX_ZERO` and also for `W1LMIN_MAX`, `W1LMIN_CNT` and `W1LMIN_ZERO`.

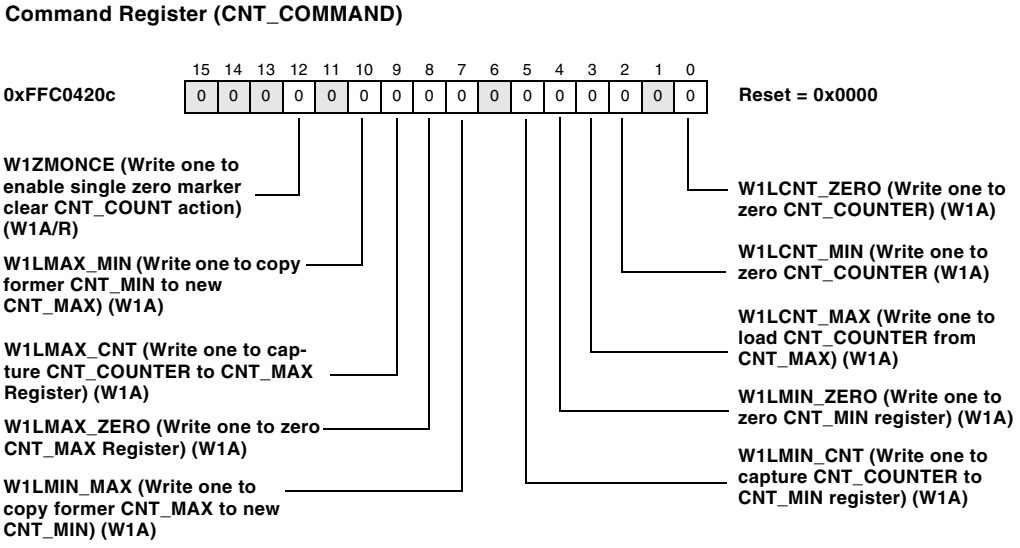


Figure 13-11. Command Register

Debounce Prescale (CNT_DEBOUNCE) Register

The debounce prescale (CNT_DEBOUNCE) register is used to select the noise filtering characteristic of the input pins (See Figure 13-12). Bits [4:0] determine the filter time. The register can be accessed at any time with 16-bit read and write operations.

$$t_{filter} = 128 \times (2^{DPRESCALE} \div SCLK)$$

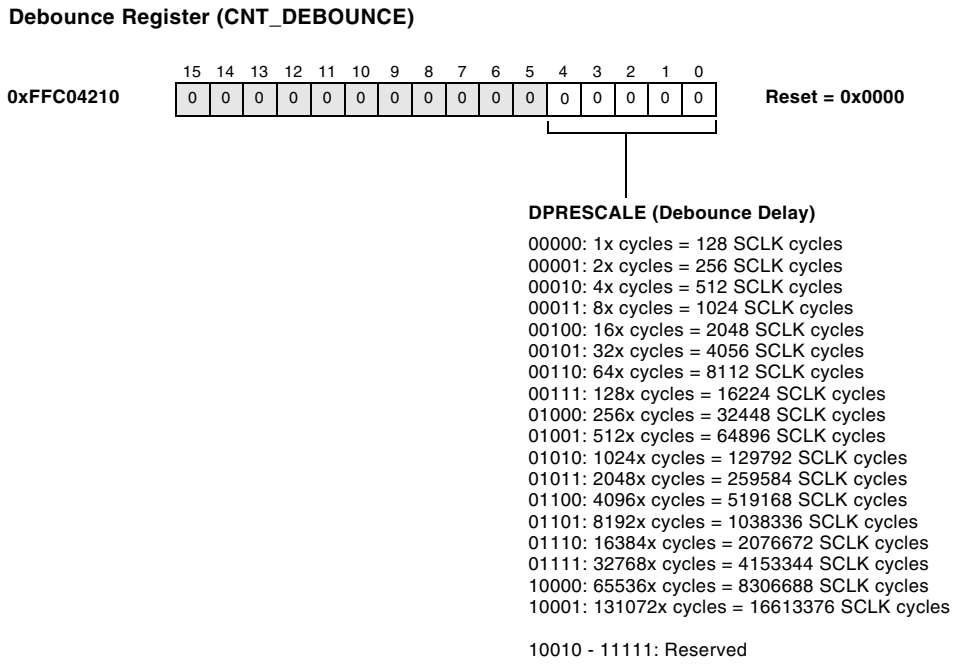


Figure 13-12. Debounce Register

Counter (CNT_COUNTER) Register

The counter (CNT_COUNTER) register holds the 32-bit, two's-complement, count value (See Figure 13-13). It can be read and written at any time. Hardware ensures that reads and writes are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows use of the rotary encoder as a 16-bit counter, if sufficient for the application.

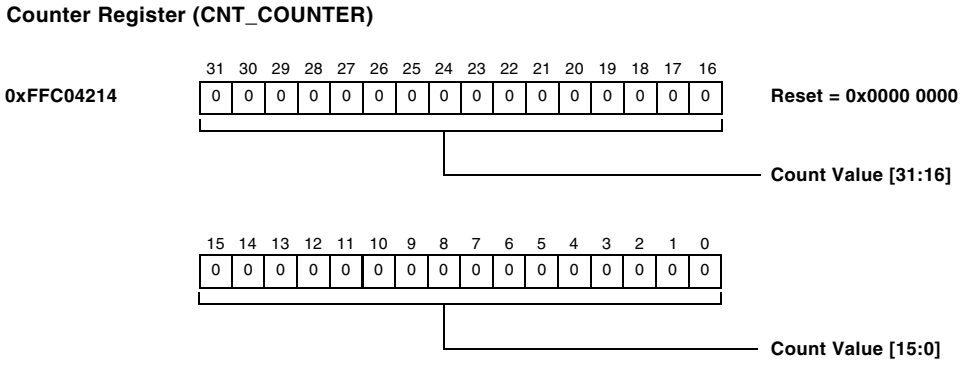


Figure 13-13. Counter Register

Boundary (CNT_MIN and CNT_MAX) Registers

The boundary (`CNT_MIN` and `CNT_MAX`) registers hold the 32-bit, two's-complement, lower and upper boundary values (See Figure 13-14 and Figure 13-15). They can be read from and written to at any time. Hardware ensures that reads and writes are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows for using the rotary encoder as a 16-bit counter if sufficient for the application.

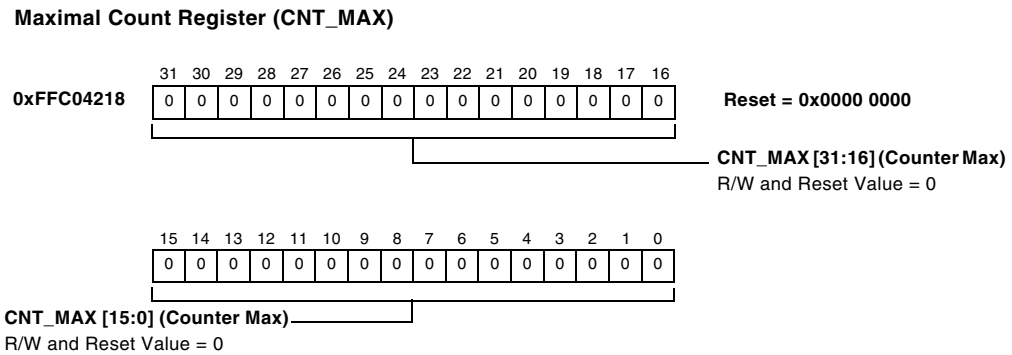


Figure 13-14. Maximal Count Register

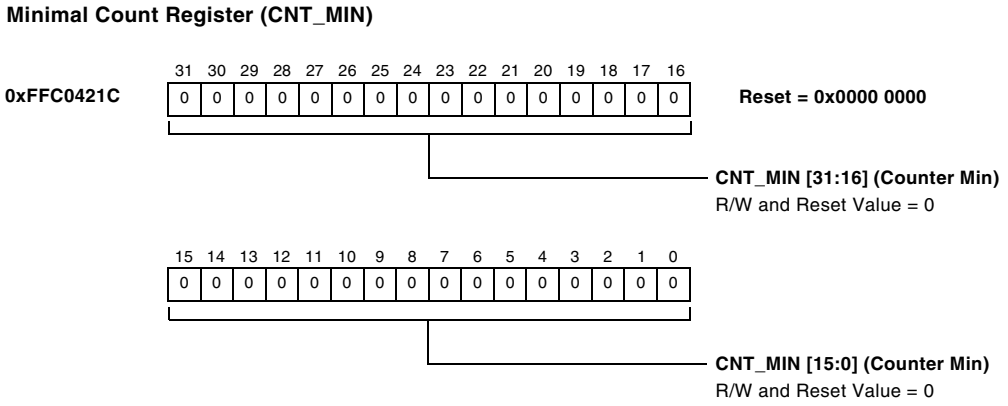


Figure 13-15. Minimal Count Register

Programming Examples

Listing 13-1 illustrates how to configure the port registers to enable rotary counter functionality through the PORTx_MUX and PORTx_FER registers.

Listing 13-1. Configuring the Port Registers to Enable Rotary Counter

```
/* enable CDG and CUD features. */
P5.H = hi(PORTH_FER);
P5.L = lo(PORTH_FER);
R5.L = nPH15 | nPH14 | nPH13 | nPH12 | PH11 | nPH10 | nPH9 | nPH8
       | nPH7 | nPH6 | nPH5 | PH4 | PH3 | nPH2 | nPH1 | nPH0;
w[P5] = R5.L;

/* enable CZM feature. */
P5.H = hi(PORTG_FER);
P5.L = lo(PORTG_FER);
R5.L = nPG15 | nPG14 | nPG13 | nPG12 | PG11 | nPG10 | nPG9 | nPG8
       | nPG7 | nPG6 | nPG5 | nPG4 | PG3 | nPG2 | nPG1 | nPG0;
```

```
w[P5] = R5.L;

/* enable CDG and CUD MUX mode. */
P5.H = hi(PORTH_MUX);
P5.L = lo(PORTH_MUX);
R5.H = hi(MUX(0,0,0,0,0,0,0,0,0,0,0,2,2,0,0,0));
R5.L = lo(MUX(0,0,0,0,0,0,0,0,0,0,0,2,2,0,0,0));
[P5] = R5;

/* enable CZM MUX mode. */
P5.H = hi(PORTG_MUX);
P5.L = lo(PORTG_MUX);
R5.H = hi(MUX(0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0));
R5.L = lo(MUX(0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0));
[P5] = R5;
```

Listing 13-2 illustrates how to initialize the rotary counter for various modes. The required rotary counter interrupts are first unmasked. The rotary counter is then configured for the required mode of operation. Note at this point we do not enable the rotary counter. Finally, some GP counter MMRs are cleared, as well as any interrupts that may be pending in the `CNT_STATUS` register.

Listing 13-2. Initializing the Rotary Counter

```
/* Setup Counter Interrupts */
P5.H = hi(CNT_IMASK);
P5.L = lo(CNT_IMASK);
R5 = nCZMZIE   /* Counter zeroed by zero marker interrupt */
    | CZMEIE   /* Zero marker error interrupt */
    | CZMIE    /* CZM pin interrupt (pushbutton) */
    | CZEROIE /* Counts to zero interrupt */
    | nCOV15IE /* Counter bit 15 overflow interrupt */
    | nCOV31IE /* Counter bit 31 overflow interrupt */
```

```
    | MAXCIE   /* Max count interrupt */
    | MINCIE   /* Min count interrupt */
    | DCIE     /* Downcount interrupt */
    | UCIE     /* Upcount interrupt */
    | ICIE (z);    /* Illegal gray/binary code interrupt */
w[P5] = R5;

/* Configure the Rotary Counter mode of operation */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = nINPDIS           /* Enable CUD and CDG inputs */
    | BNDMODE_COMP     /* Boundary compare mode */
    | nZMZC            /* Disable Zero Counter Enable */
    | CNTMODE_QUADENC /* Quadrature Encoder Mode */
    | CZMINV           /* Polarity of CZM pin */
    | nCUDINV          /* Polarity of CUD pin */
    | nCDGINV          /* Polarity of CDG Pin */
    | nDEBE            /* Disable the debounce */
    | nCNTE (z);       /* Disable the counter */
w[P5] = R5;

/* Zero the CNT_COUNT, CNT_MIN and CNT_MAX registers
This is optional as after reset they are default to zero */
P5.H = hi(CNT_COMMAND);
P5.L = lo(CNT_COMMAND);
R5 = W1LCNT_ZERO | W1LMIN_ZERO | W1LMAX_ZERO (z);
w[P5] = R5;

/* Clear any identified interrupts */
P5.H = hi(CNT_STATUS);
P5.L = lo(CNT_STATUS);
R5.L = ICII     /* Illegal Gray/Binary Code Interrupt Identifier
*/
       | UCII   /* Up count Interrupt Identifier */
```

```
        | DCII   /* Down count Interrupt Identifier */
        | MINCII /* Min Count Interrupt Identifier */
        | MAXCII /* Max Count Interrupt Identifier */
        | COV31II /* Bit 31 Overflow Interrupt Identifier */
        | COV15II /* Bit 15 Overflow Interrupt Identifier */
        | CZEROII /* Count to Zero Interrupt Identifier */
        | CZMII   /* CZM Pin Interrupt Identifier */
        | CZMEII /* CZM Error Interrupt Identifier */
        | CZMZII; /* CZM Zeroes Counter Interrupt Identifier */
w[P5] = R5;
```

Listing 13-3 illustrates how to set up the peripheral and core interrupts for the rotary counter. The counter interrupts generated on `IRQ68` are mapped to the `IVG7` interrupt. Finally the system and peripheral interrupts are unmasked and then the rotary counter is enabled.

Listing 13-3. Setting Up the Interrupts for the Rotary Counter

```
/* Assign the CNT interrupt to IVG7 */
P5.H = hi(SIC_IAR8);
P5.L = lo(SIC_IAR8);
R6.H = hi(0xFFF0FFFF);
R6.L = lo(0xFFF0FFFF);
R7.H = hi(0x00000000);
R7.L = lo(0x00000000);
R5 = [P5];
R5 = R5 & R6; /* zero the Counter interrupt field */
R5 = R5 | R7; /* set Counter interrupt to required priority */
[P5] = R5;

/* Set up the interrupt vector for the rotary counter */
R5.H = hi(_IVG7_handler);
R5.L = lo(_IVG7_handler);
P5.H = hi(EVT7);
```

```
P5.L = lo(EVT7);
[P5] = R5;

/* Unmask IVG7 interrupt in the IMASK register */
P5.H = hi(IMASK);
P5.L = lo(IMASK);
R5 = [P5];
bitset(R5, bitpos(EVT_IVG7));
[P5] = R5;

/* Unmask interrupt 68 generated by the counter */
P5.H = hi(SIC_IMASK2);
P5.L = lo(SIC_IMASK2);
R5 = [P5];
bitset(R5, bitpos(IRQ_CNT));
[P5] = R5;

/* Enable the Rotary Counter */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = w[P5](z);
bitset(R5, bitpos(CNTE));
w[P5] = R5.L;
```

Listing 13-4 illustrates a sample interrupt handler that is responsible for servicing the rotary counter interrupts. On entry to the handler, the `SIC_ISR2` register is interrogated to determine if the counter is waiting for a service interrupt. If a counter interrupt is waiting to be serviced, then the handler that is responsible for processing all counter interrupts is called.

Listing 13-4. Sample Interrupt Handler Rotary Counter Interrupts

```
_IVG7_handler:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = ASTAT;
    [--SP] = (R7:0, P5:0);

    /* Was it a counter interrupt? */
    P5.H = hi(SIC_ISR2);
    P5.L = lo(SIC_ISR2);
    R5 = [P5];
    CC = bittst(R5, bitpos(IRQ_CNT));
    IF !CC JUMP _IVG7_handler.completed;
    CALL _IVG7_handler.counter;

    _IVG7_handler.completed:

    SSYNC;
    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    ASTAT = [SP++];
    RETS = [SP++];
    RTI;    /* Exit the interrupt service routine */
_IVG7_handler.end:

_IVG7_handler.counter:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = (R7:0, P5:0);

    /* Determine what counter interrupts we wish to service */
    P5.H = hi(CNT_IMASK);
    P5.L = lo(CNT_IMASK);
```

```
    R5 = w[P5](z);

    P5.H = hi(CNT_STATUS);
    P5.L = lo(CNT_STATUS);
    R6 = w[P5](z);
    R5 = R5 & R6;

/* Interrupt handlers for all rotary counter interrupts */
_IVG7_handler.counter.illegal_code:
    CC = bittst(R5, bitpos(ICII));
    IF !CC JUMP _IVG7_handler.counter.up_count;

    /* Clear the serviced request */
    R6 = ICII (z);
    w[P5] = R6;

    /* insert illegal code handler here */

_IVG7_handler.counter.illegal_code.end:


_IVG7_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG7_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

_IVG7_handler.counter.up_count.end:

_IVG7_handler.counter.down_count:
```

```
    CC = bittst(R5, bitpos(DCII));
    IF !CC JUMP _IVG7_handler.counter.min_count;

    /* Clear the serviced request */
    R6 = DCII (z);
    w[P5] = R6;

    /* insert down count handler here */

_IVG7_handler.counter.down_count.end:

_IVG7_handler.counter.min_count:
    CC = bittst(R5, bitpos(MINCII));
    IF !CC JUMP _IVG7_handler.counter.max_count;

    /* Clear the serviced request */
    R6 = MINCII (z);
    w[P5] = R6;

    /* insert min count handler here */

_IVG7_handler.counter.min_count.end:


_IVG7_handler.counter.max_count:
    CC = bittst(R5, bitpos(MAXCII));
    IF !CC JUMP _IVG7_handler.counter.b31_overflow;

    /* Clear the serviced request */
    R6 = MAXCII (z);
    w[P5] = R6;

    /* insert max count handler here */
```

```
_IVG7_handler.counter.max_count.end:

_IVG7_handler.counter.b31_overflow:
    CC = bittst(R5, bitpos(COV31II));
    IF !CC JUMP _IVG7_handler.counter.b15_overflow;

    /* Clear the serviced request */
    R6 = COV31II (z);
    w[P5] = R6;

    /* insert bit 31 overflow handler here */

_IVG7_handler.counter.b31_overflow.end:

_IVG7_handler.counter.b15_overflow:
    CC = bittst(R5, bitpos(COV15II));
    IF !CC JUMP _IVG7_handler.counter.count_to_zero;

    /* Clear the serviced request */
    R6 = COV15II (z);
    w[P5] = R6;

    /* insert bit 15 overflow handler here */

_IVG7_handler.counter.b15_overflow.end:

_IVG7_handler.counter.count_to_zero:
    CC = bittst(R5, bitpos(CZEROII));
    IF !CC JUMP _IVG7_handler.counter.czm;

    /* Clear the serviced request */
    R6 = CZEROII (z);
    w[P5] = R6;
```

```
    /* insert count to zero handler here */

_IVG7_handler.counter.count_to_zero.end:

_IVG7_handler.counter.czm:
    CC = bittst(R5, bitpos(CZMII));
    IF !CC JUMP _IVG7_handler.counter.czm_error;

    /* Clear the serviced request */
    R6 = CZMII (z);
    w[P5] = R6;

    /* insert czm handler here */

_IVG7_handler.counter.czm.end:

_IVG7_handler.counter.czm_error:
    CC = bittst(R5, bitpos(CZMEII));
    IF !CC JUMP _IVG7_handler.counter.czm_zeroes_counter;

    /* Clear the serviced request */
    R6 = CZMEII (z);
    w[P5] = R6;

    /* insert czm error handler here */

_IVG7_handler.counter.czm_error.end:


_IVG7_handler.counter.czm_zeroes_counter:
    CC = bittst(R5, bitpos(CZMZII));
    IF !CC JUMP _IVG7_handler.counter.all_serviced;

    /* Clear the serviced request */
```

```
    R6 = CZMZII (z);
    w[P5] = R6;

    /* insert czm zeroes counter handler here */

_IVG7_handler.counter.czm_zeroes_counter.end:

_IVG7_handler.counter.all_serviced:

    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    RETS = [SP++];
    RTS;
_IVG7_handler.counter.end:
```

Listing 13-5 illustrates how to set up timer 6 in order to capture the period of counter events. The timer is configured for `WDTH_CAP` mode and the period between the last two successive counter events is read from within the up count interrupt handler that was provided in Listing 13-4.

Listing 13-5. Setting Up Timer 6 for Counter Event Period Capture

```
/* configure the timer for WDTH_CAP mode */
P5.H = hi(TIMER6_CONFIG);
P5.l = lo(TIMER6_CONFIG);
R5 = PULSE_HI | PERIOD_CNT | TIN_SEL | WDTH_CAP (z);
w[P5] = R5.l;

/* Enable Timer 6 */
P5.H = hi(TIMER_ENABLE0);
P5.L = lo(TIMER_ENABLE0);
R5 = TIMEN6 (z);
w[P5] = R5.L;

...
```

```
_IVG7_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG7_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

    /* Read the period between the last two successive events */
    P5.H = hi(TIMER6_PERIOD);
    P5.L = lo(TIMER6_PERIOD);
    R5 = [P5];

    P5.H = hi(_event_period);
    P5.L = lo(_event_period);
    [P5] = R5;
_IVG7_handler.counter.up_count.end:
```

14 REAL-TIME CLOCK

This chapter describes the real-time clock (RTC) and includes the following sections:

- "Overview" on page 14-1
- "Interface Overview" on page 14-3
- "Description of Operation" on page 14-3
- "RTC Programming Model" on page 14-6
- "RTC Registers" on page 14-20
- "Programming Examples" on page 14-24

Overview

The RTC provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter, which counts the elapsed time since the last system reset.

The RTC watch features are clocked by a 32.768 kHz crystal external to the processor. The RTC uses dedicated power supply pins and is independent of any reset, which enables it to maintain functionality even when the rest of the processor is powered down.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60-second counter
- 60-minute counter
- 24-hour counter
- 32768-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 32768-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.

The RTC provides two alarm features, programmed with the RTC alarm register (`RTC_ALARM`). The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC stopwatch count register (`RTC_SWCNT`). When the stopwatch interrupt is enabled and the specified number of seconds has elapsed, the RTC generates an interrupt.

Interface Overview

The RTC external interface consists of two clock pins, which together with the external components form the reference clock circuit for the RTC. The RTC interfaces internally to the processor system through the peripheral access bus (PAB), and through the interrupt interface to the SIC (system interrupt controller).

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off. Figure 14-1 on page 14-4 provides a block diagram of the RTC.

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.

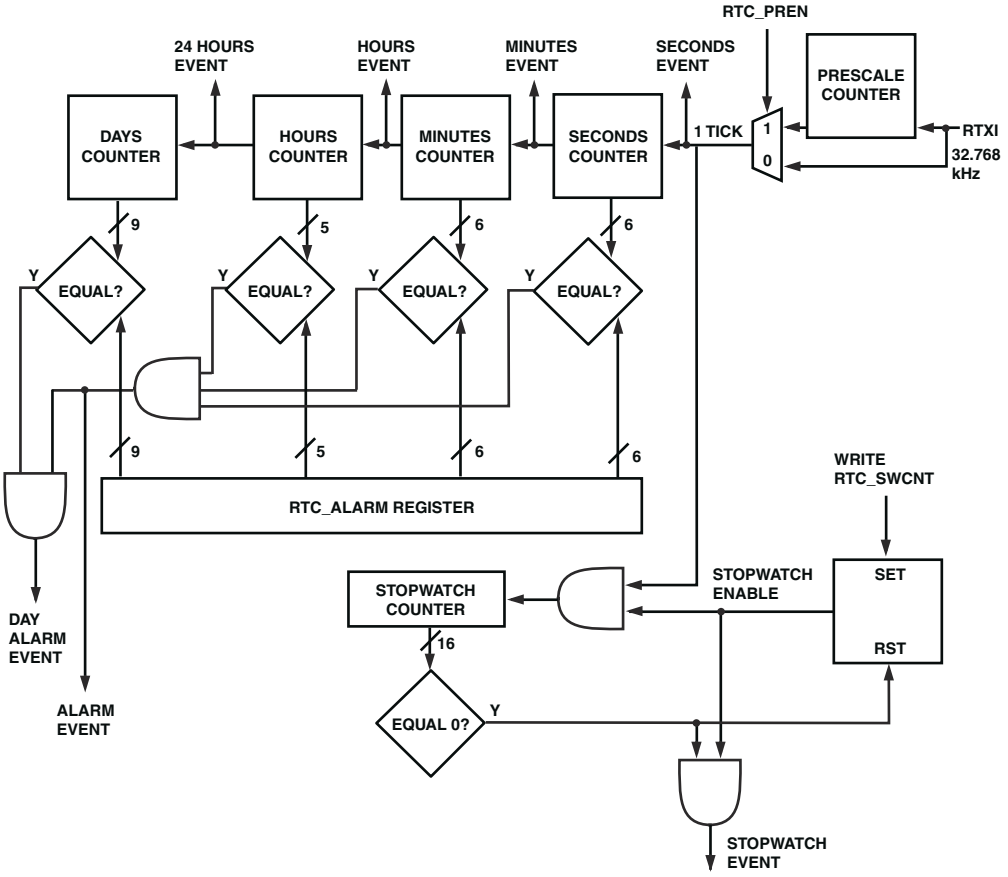


Figure 14-1. RTC Block Diagram

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 through RTC_PREN and do not dynamically alter the state of the prescaler during normal operation.

Running without the prescaler enabled is provided primarily as a test mode. All functionality works, just 32,768 times as fast. Typical software should never program RTC_PREN to 0. The only reason to do so is to synchronize the 1 Hz tick to a more precise external event, as the 1 Hz tick predictably occurs a few RTXI cycles after a 0-to-1 transition of RTC_PREN.

Use the following sequence to achieve synchronization to within 100 ms.

1. Write RTC_PREN to 0.
2. Wait for the write to complete.
3. Wait for the external event.
4. Write RTC_PREN to 1.
5. Wait for the write to complete.
6. Reprogram the time into RTC_STAT.

RTC Programming Model

The RTC programming model consists of a set of system MMRs. Software can configure the RTC and can determine the status of the RTC through reads and writes to these registers. The RTC interrupt control register (RTC_ICTL) and the RTC interrupt status register (RTC_ISTAT) provide RTC interrupt management capability.

Note that software cannot disable the RTC counting function. However, all RTC interrupts can be disabled, or masked. At reset, all interrupts are disabled. The RTC state can be read through the system MMR status registers at any time.

The primary RTC functionality, shown in Figure 14-1 on page 14-4, consists of registers and counters that are powered by an independent RTC V_{DD} supply. This logic is never reset; it comes up in an unknown state when RTC V_{DD} is first powered on.

The RTC also contains logic powered by the same internal V_{DD} as the processor core and other peripherals. This logic contains some control functionality, holding registers for PAB write data, and prefetched PAB read data shadow registers for each of the five RTC V_{DD}-powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

Figure 14-2 shows the connections between the RTC V_{DD}-powered RTC MMRs and their corresponding internal V_{DD}-powered write holding registers and read shadow registers. In the figure, “REG” means each of the RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN registers. The RTC_ISTAT register connects only to the PAB.

The rising edge of the 1 Hz RTC clock is the “1 Hz tick”. Software can synchronize to the 1 Hz tick by waiting for the seconds event flag to set or by waiting for the seconds interrupt (if enabled).

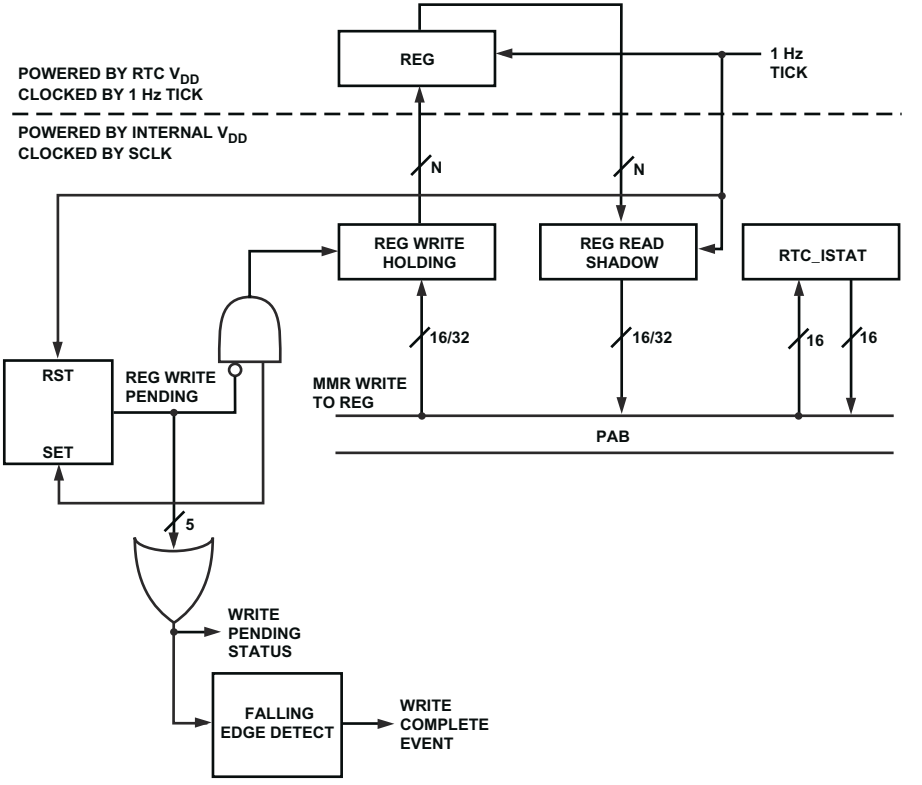


Figure 14-2. RTC Register Architecture

Register Writes

Writes to all RTC MMRs, except the RTC interrupt status register (RTC_ISTAT), are saved in write holding registers and then are synchronized to the RTC 1 Hz clock. The write pending status bit in RTC_ISTAT indicates the progress of the write. The write pending status bit is set when a write is initiated and is cleared when all writes are complete. The falling edge of the write pending status bit causes the write complete flag in RTC_ISTAT to be set. This flag can be configured in RTC_ICTL to cause an interrupt. Software does not have to wait for writes to an RTC MMR to

complete before writing to another RTC MMR. The write pending status bit is set if any writes are in progress, and the write complete flag is set only when all writes are complete.

Any writes in progress when peripherals are reset are aborted. Do not stop SCLK (enter deep sleep mode) or remove Internal V_{DD} power until all RTC writes have completed.

Do not attempt another write to the same register without waiting for the previous write to complete. Subsequent writes to the same register are ignored if the previous write is not complete.

Reading a register that is written before the write complete flag is set returns the old value. Always check the write pending status bit before attempting a read or write.

Write Latency

Writes to the RTC MMRs are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the RTC is to monitor the seconds (1 Hz) event flag or to program an interrupt for this event and then write the current time to the RTC status register (RTC_STAT) in the interrupt service routine (ISR). The new value is inserted ahead of the incrementer. Hardware adds one second to the written value (with appropriate carries into minutes, hours and days) and loads the incremented value at the next 1 Hz tick, when it represents the then-current time.

Writes posted at any time are properly synchronized to the 1 Hz clock. Writes complete at the rising edge of the 1 Hz clock. A write posted just before the 1 Hz tick may not be completed until the 1 Hz tick one second later. Any write posted in the first 990 µs 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. During deep sleep, the internal V_{DD} shadow registers are not updated, but neither can they be read.

During deep sleep state, all bits in RTC_ISTAT are cleared. Events that occur during deep sleep are not recorded in RTC_ISTAT. The internal V_{DD} RTC control logic generates a virtual 1 Hz tick within one RTXI period (30.52 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 write-1-to-clear the W1C bits in RTC_ISTAT. All W1C bits are already cleared by hardware. The seconds event flag is set when the RTC internal V_{DD} logic has completed its restart sequence. Software should wait until the seconds event flag is set and then may begin reading or writing any RTC register.

Event Flags

The unknown values in the registers at powerup can cause event flags to set before the correct value is written into each of the registers. By catching the 1 Hz clock edge, the write to RTC_STAT can occur a full second before the write to RTC_ALARM. This would cause an extra second of delay between the validity of RTC_STAT and RTC_ALARM, if the value of the RTC_ALARM out of reset is the same as the value written to RTC_STAT. Wait for the writes to complete on these registers before using the flags and interrupts associated with their values.

The following is a list of flags along with the conditions under which they are valid:

- Seconds (1 Hz) event flag

  Always set on the positive edge of the 1 Hz clock and after shadow registers have updated after waking from deep sleep. This is valid as long as the RTC 1 Hz clock is running. Use this flag or interrupt to validate the other flags.

- Write complete and write pending status

  Always valid.

- Minutes event flag

  Valid only after the second field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- Hours event flag

  Valid only after the minute field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- 24 Hours event flag

  Valid only after the hour field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- Stopwatch event flag

  Valid only after the RTC_SWCNT register is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_SWCNT value before using this flag value or enabling the interrupt.

- Alarm event and day alarm event flags

  Valid only after the RTC_STAT and RTC_ALARM registers are valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT and RTC_ALARM values before using this flag value or enabling its interrupt.

Writes posted together at the beginning of the same second take effect together at the next 1 Hz tick. The following sequence is safe and does not result in any spurious interrupts from a previous state.

1. Wait for 1 Hz tick.
2. Write-1-to-clear the RTC_ISTAT flags for alarm, day alarm, stopwatch, and/or per-interval.
3. Write new values for RTC_STAT, RTC_ALARM, and/or RTC_SWCNT.
4. Write new value for RTC_ICTL with alarm, day alarm, stopwatch, and/or per-interval interrupts enabled.
5. Wait for 1 Hz tick.
6. New values have now taken effect simultaneously.

Setting Time of Day

The RTC status register (RTC_STAT) is used to read or write the current time. Reads return a 32-bit value that always reflects the current state of the days, hours, minutes, and seconds counters. Reads and writes must be 32-bit transactions; attempted 16-bit transactions result in an MMR error. Reads always return a coherent 32-bit value. The hours, minutes, and seconds fields are usually set to match the real time of day. The day counter value is incremented every day at midnight to record how many days have elapsed since it was last modified. Its value does not correspond to a particular calendar day. The 15-bit day counter provides a range of 89 years, 260 or 261 days (depending on leap years) before it overflows.

After the 1 Hz tick, program RTC_STAT with the current time. At the next 1 Hz tick, RTC_STAT takes on the new, incremented value. For example:

1. Wait for 1 Hz tick.
2. Read RTC_STAT, get 10:45:30.

3. Write RTC_STAT to current time, 13:10:59.
4. Read RTC_STAT, still get old time 10:45:30.
5. Wait for 1 Hz tick.
6. Read RTC_STAT, get new current time, 13:11:00.

Using the Stopwatch

The RTC stopwatch count (RTC_SWCNT) register contains the countdown value for the stopwatch. The stopwatch counts down seconds from the programmed value and generates an interrupt (if enabled) when the count reaches 0. The counter stops counting at this point and does not resume counting until a new value is written to RTC_SWCNT. Once running, the counter may be overwritten with a new value. This allows the stopwatch to be used as a watchdog timer with a precision of one second.

The stopwatch can be programmed to any value between 0 and $(2^{16} – 1)$ seconds, which is a range of 18 hours, 12 minutes, and 15 seconds.

Typically, software should wait for a 1 Hz tick, then write RTC_SWCNT. One second later, RTC_SWCNT changes to the new value and begins decrementing. Because the register write occupies nearly one second, the time from writing a value of N until the stopwatch interrupt is nearly N + 1 seconds. To produce an exact delay, software can compensate by writing N – 1 to get a delay of nearly N seconds. This implies that you cannot achieve a delay of 1 second with the stopwatch. Writing a value of 1 immediately after a 1 Hz tick results in a stopwatch interrupt nearly two seconds later. To wait one second, software should just wait for the next 1 Hz tick.

The RTC_SWCNT register is not reset. After initial powerup, it may be running. When the stopwatch is not used, writing it to 0 to force it to stop saves a small amount of power.

Interrupts

The RTC can provide interrupts at several programmable intervals:

- Per second, minute, hour, and day—based on increments to the respective counters in RTC_STAT
- On countdown from a programmable value—value in RTC_SWCNT transitions to 0 or is written with 0 by software (whether it was previously running or already stopped with a count of 0)
- Daily at a specific time—all fields of RTC_ALARM must match RTC_STAT except the day field
- On a specific day and time—all fields of RTC_ALARM register must match RTC_STAT

The RTC can be programmed to provide an interrupt at the completion of all pending writes to any of the 1 Hz registers (RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN). The eight RTC interrupt events can be individually masked or enabled by the RTC interrupt control register (RTC_ICTL). The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24-hour interrupt occurs once per 24-hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wakeup request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC interrupt and the RTC wakeup to the PLL are handled specially and are masked (forced low) until after the first write to the RTC_ICTL register is complete. Therefore, all interrupts act as if they were disabled at system reset (as if all bits of RTC_ICTL were zero), even though some bits of RTC_ICTL may read as nonzero. If no RTC

interrupts are needed immediately after reset, it is recommended to write RTC_ICTL to 0x0000 so that later read-modify-write accesses function as intended.

Interrupt status can be determined by reading the RTC interrupt status register (RTC_ISTAT). All bits in RTC_ISTAT are sticky. Once set by the corresponding event, each bit remains set until cleared by a software write to this register. Event flags are always set; they are not masked by the interrupt enable bits in RTC_ICTL. Values are cleared by writing a 1 to the respective bit location, except for the write pending status bit, which is read-only. Writes of 0 to any bit of the register have no effect. This register is cleared at reset and during deep sleep.

The RTC interrupt is set whenever an event latched into the RTC_ISTAT register is enabled in the RTC_ICTL register. The pending RTC interrupt is cleared whenever all enabled and set bits in RTC_ISTAT are cleared, or when all bits in RTC_ICTL corresponding to pending events are cleared.

As shown in Figure 14-3, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wake-up from a sleep state. The RTC generates a separate signal for wake-up from a deep sleep or from an internal V_{DD} power-off state. The deep sleep wake-up signal is asserted at the 1 Hz tick when any RTC interval event enabled in RTC_ICTL occurs. The assertion of the deep sleep wake-up signal causes the processor core clock (CCLK) and the system clock (SCLK) to restart. Any enabled event that asserts the RTC deep sleep wake-up signal also causes the RTC IRQ to assert once SCLK restarts.

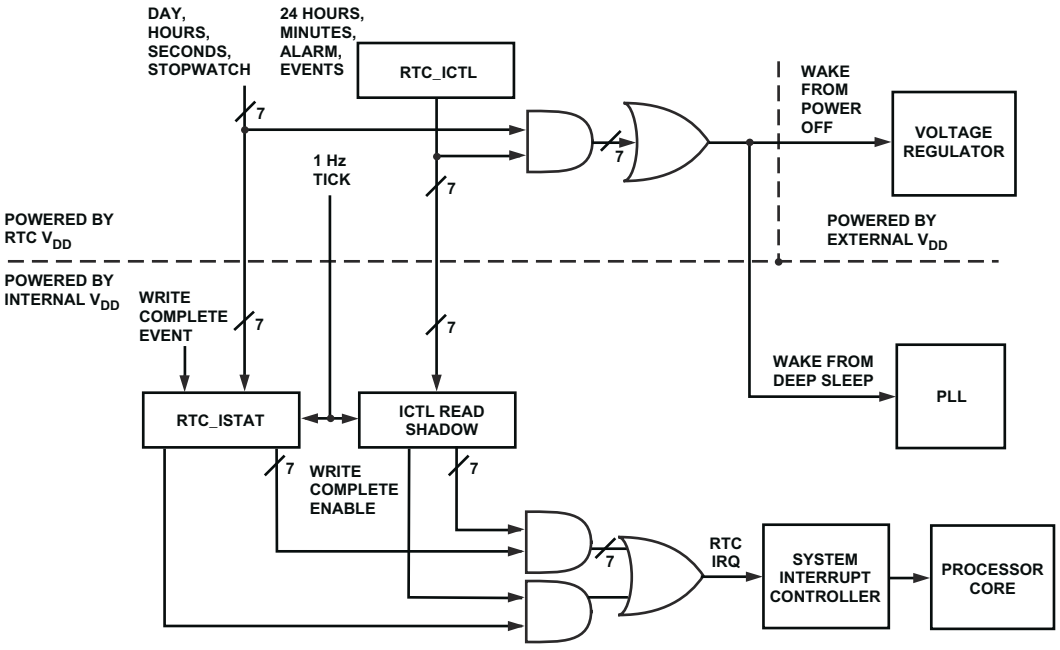


Figure 14-3. RTC Interrupt Structure

State Transitions Summary

Table 14-1 shows how each RTC MMR is affected by the system states. The phase-locked loop (PLL) states (reset, full on, active, sleep, and deep sleep) are defined in Chapter 18, "Dynamic Power Management". "No power" means none of the processor power supply pins are connected to a source of energy. "Off" means the processor core, peripherals, and memory are not powered (internal V_{DD} is off), while the RTC is still powered and running. External V_{DD} may still be powered. Registers described as "as written" are holding the last value software wrote to the register. If the register has not been written since RTC V_{DD} power was applied, then the state is unknown (for all bits of RTC_STAT, RTC_ALARM, and RTC_SWCNT, and for some bits of RTC_ISTAT, RTC_PREN, and RTC_ICTL).

Table 14-1. Effect of States on RTC MMRs

RTC V_{DD}	IV_{DD}	System State	RTC_ICTL	RTC_ISTAT	RTC_STAT RTC_SWCNT	RTC_ALARM RTC_PREN
Off	Off	No power	X	X	X	X
On	On	Reset	As written	0	Counting	As written
On	On	Full on	As written	Events	Counting	As written
On	On	Sleep	As written	Events	Counting	As written
On	On	Active	As written	Events	Counting	As written
On	On	Deep sleep	As written	0	Counting	As written
On	Off	Off	As written	X	Counting	As written

Table 14-2 summarizes software's responsibilities with respect to the RTC at various system state transition events.

Table 14-2. RTC System State Transition Events

At This Event:	Execute This Sequence:
Power on from no power	Write RTC_PREN = 1. Wait for write complete. Write RTC_STAT to current time. Write RTC_ALARM, if needed. Write RTC_SWCNT. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.
Full on after reset or Full on after power on from off	Wait for seconds event, or write RTC_PREN = 1 and wait for write complete. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts. Read RTC MMRs as required.

Table 14-2. RTC System State Transition Events (Cont'd)

At This Event:	Execute This Sequence:
Wake from deep sleep	Wait for seconds event flag to set. Write RTC_ISTAT to acknowledge RTC deep sleep wakeup. Read RTC MMRs as required. The PLL state is now active. Transition to full on as needed.
Wake from sleep	If wakeup came from RTC, seconds event flag is set. In this case, write RTC_ISTAT to acknowledge RTC wakeup IRQ. Always, read RTC MMRs as required.
Before going to sleep	If wakeup by RTC is desired: Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC interrupt sources for wakeup. Wait for write complete. Enable RTC for wakeup in the system interrupt wakeup-enable register (SIC_IWR).
Before going to deep sleep	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC event sources for deep sleep wakeup. Wait for write complete.
Before going to off	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable any desired RTC event sources for powerup wakeup. Wait for write complete. Set the wake bit in the voltage regulator control register (VR_CTL).

RTC Registers

The following sections provide register definitions. Illustrations are shown in Figure 14-4 through Figure 14-9.

Table 14-3 shows the functions of the RTC registers.

Table 14-3. RTC Register Mapping

Register Name	For More Info	Notes
RTC_STAT	"RTC Status (RTC_STAT) Register" on page 14-21	Holds time of day
RTC_ICTL	"RTC Interrupt Control (RTC_ICTL) Register" on page 14-21	Bits 14:7 are reserved
RTC_ISTAT	"RTC Interrupt Status (RTC_ISTAT) Register" on page 14-22	Bits 13:7 are reserved
RTC_SWCNT	"RTC Stopwatch Count (RTC_SWCNT) Register" on page 14-22	Undefined at reset
RTC_ALARM	"RTC Alarm (RTC_ALARM) Register" on page 14-23	Undefined at reset
RTC_PREN	"RTC Prescaler Enable (RTC_PREN) Register" on page 14-23	Always set PREN = 1 for 1 Hz ticks

RTC Status (RTC_STAT) Register

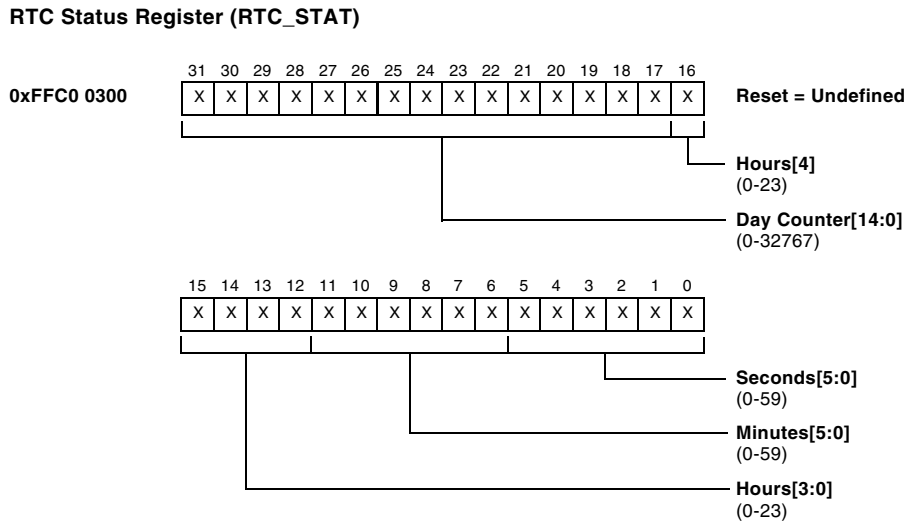


Figure 14-4. RTC Status Register

RTC Interrupt Control (RTC_ICTL) Register

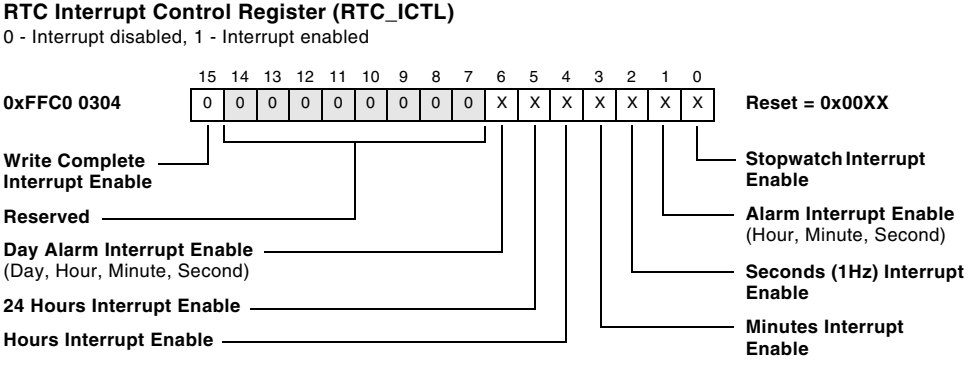


Figure 14-5. RTC Interrupt Control Register

RTC Interrupt Status (RTC_ISTAT) Register

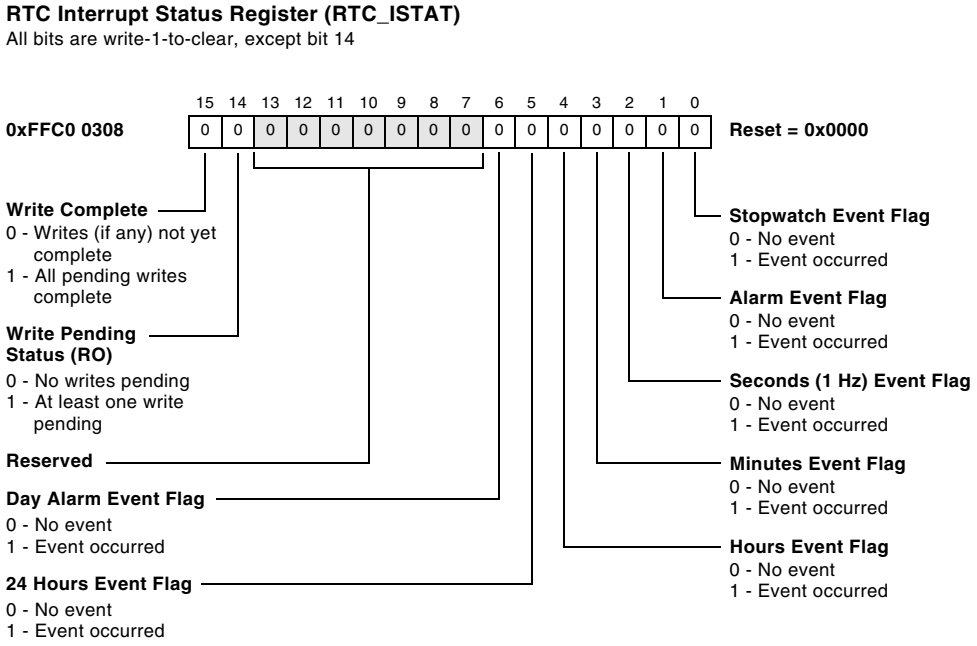


Figure 14-6. RTC Interrupt Status Register

RTC Stopwatch Count (RTC_SWCNT) Register

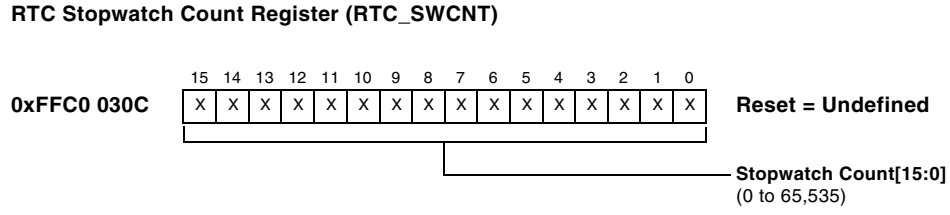


Figure 14-7. RTC Stopwatch Count Register

RTC Alarm (RTC_ALARM) Register

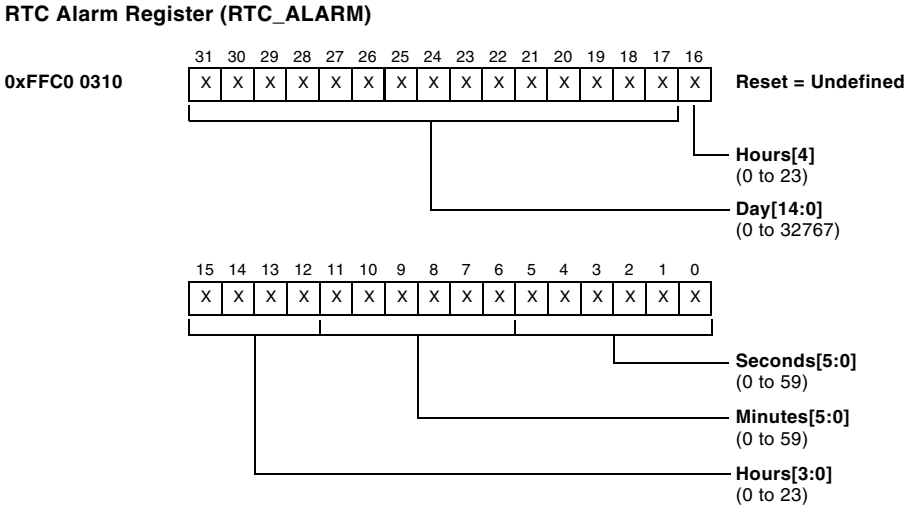


Figure 14-8. RTC Alarm Register

RTC Prescaler Enable (RTC_PREN) Register

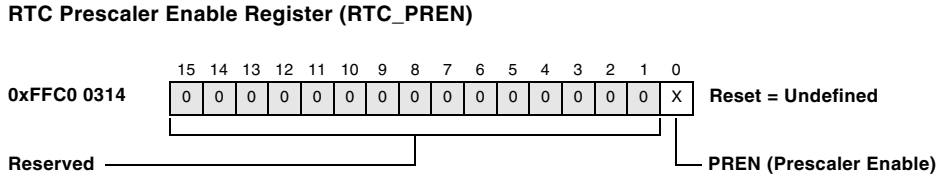


Figure 14-9. Prescaler Enable Register

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 <defBF54x.h>` for ADSP-BF54x processor projects).

Enable RTC Prescaler

Listing 14-1 properly enables the prescaler and clears any pending interrupts.

Listing 14-1. Enabling the RTC Prescaler

```
RTC_Initialization:
  P0.H = HI(RTC_PREN);
  P0.L = LO(RTC_PREN);
  R0=PREN(Z);  /* enable pre-scalar for 1 Hz ticks */
  W[P0] = R0.L;

  P0.L = LO(RTC_ISTAT);
  R0 = 0x807F(Z);
  W[P0] = R0.L;   /* clear any pending interrupts */

  R0 = WRITE_COMPLETE(Z);  /* mask for WRITE-COMPLETE bit */
  Poll_WC:   R1 = W[P0](Z);
             R1 = R1 & R0;  /* wait for Write Complete */
             CC = AZ;
             IF CC JUMP Poll_WC;
  RTS;
```

RTC Stopwatch For Exiting Deep Sleep Mode

Listing 14-2 sets up the RTC to utilize the stopwatch feature to come out of deep sleep mode. This code assumes that the `_RTC_Interrupt` is properly registered as the ISR for the real-time clock event, the RTC interrupt is enabled in both `IMASK` and `SIC_IMASK`, and that the RTC prescaler has already been enabled properly.

Listing 14-2. RTC Stopwatch Interrupt to Exit Deep Sleep

```
/* RTC Wake-Up Interrupt To Be Used With Deep Sleep Code */
_RTC_Interrupt:
  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 14-3 sets up the RTC to utilize the alarm feature to come out of hibernate state. This code assumes that the prescaler has already been properly enabled.

Listing 14-3. Setting RTC Alarm to Exit Hibernate State

```
Hibernate_Code:
  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 */
```

15 ENHANCED PARALLEL PERIPHERAL INTERFACE

This chapter describes the enhanced parallel peripheral interface (EPPI) and includes the following sections:

- "Overview" on page 15-1
- "Interface Overview" on page 15-4
- "Description of Operation" on page 15-6
- "Functional Description" on page 15-10
- "EPPI Data Path Options" on page 15-27
- "Programming Model" on page 15-64
- "EPPI Registers" on page 15-73

Overview

The ADSP-BF54x processor Blackfin processor provides up to three enhanced parallel peripheral interfaces (EPPIs), supporting data widths up to 24 bits wide. The EPPI supports direct connection to active TFT LCD, parallel A/D and D/A converters, video encoders and decoders, image sensor modules and other general-purpose peripherals.

The following features are supported in the EPPI module.

- Programmable data length: 8, 10, 12, 14, 16, 18 and 24 bits per clock cycle.
- Bidirectional and half-duplex port.
- Clock can be provided externally or can be generated internally.
- Various framed and non-framed operating modes. Frame syncs can be generated internally or can be supplied by an external device.
- Various general-purpose modes with one frame sync, two frame syncs, three frame syncs and zero frame sync modes for both receive and transmit.
- ITU-656 status word error detection and correction for ITU-656 Receive modes.
- ITU-656 preamble and status word decode.
- Three different modes for ITU-656 receive modes: active video only, vertical blanking only, and entire field.
- Horizontal and vertical windowing for GP 2 and 3 frame sync modes.
- Optional packing and unpacking of data to/from 32 bits from/to 8, 16 and 24 bits. If packing/unpacking is enabled, endianness can be altered to change the order of packing/unpacking of bytes/words.

- Optional sign extension or zero-fill for receive modes.
- During receive modes, alternate even or odd data samples can be filtered out.
- Programmable clipping of data values for 8-bit and 16-bit transmit modes.
- RGB888 can be converted to RGB666 or RGB565 for transmit modes.
- Various de-interleaving/interleaving modes for receiving/transmitting 4:2:2 YCrCb data.
- FIFO watermarks and urgent DMA features.
- Clock gating by an external device asserting the clock gating control signal.
- Configurable LCD data enable (DEN) output available on frame sync 3.

Each EPPI is a half-duplex, bidirectional port with a dedicated clock pin and three frame sync (FS) pins. Each EPPI has a DMA channel associated with it. Moreover, in some modes, an EPPI may use an additional DMA channel.

The EPPI supports direct connection to LCD panels, parallel A/D and D/A converters, video encoders and decoders, CMOS sensors and other general-purpose peripherals.

The ADSP-BF54x processor Blackfin processors feature up to three separate (but functionally identical) EPPI modules. The ADSP-BF544, ADSP-BF547, ADSP-BF548 and ADSP-BF549 processors feature three EPPIs, referred to as EPPI0, EPPI1, and EPPI2. EPPI0 is not present on the ADSP-BF542 processor.

To reduce pin count, some EPPI module pins are multiplexed with other EPPI pins and peripheral pins. (See Figure 15-8 on page 15-28 for more details.)

The maximum data widths are:

- EPPI0 supports up to 24 bits of data, 3 frame syncs and a clock.
- EPPI1 supports up to 16 bits of data, 3 frame syncs and a clock.
- EPPI2 supports up to 8 bits of data, 3 frame syncs and a clock.

For simplicity, discussions that apply to all EPPI blocks are denoted as EPPIx, which refers to any/all EPPI modules. The abbreviations RX and TX are also used in order to denote receive and transmit modes, respectively.

Interface Overview

A block diagram of the EPPI is shown in Figure 15-1.

The EPPI can be supplied with an external clock, or the clock can be generated internally and supplied to external devices. When using the internal clock, the maximum frequency possible for `EPPIx_CLK` is `SCLK/2`. When using an external clock, the maximum frequency for `EPPIx_CLK` is 75 MHz.

When using an external `EPPIx_CLK`, there may be up to two cycles latency before valid data is received or transmitted.

The internal clock can be generated from `SCLK` if the `ICLKGEN` bit in the `EPPIx_CONTROL` register is set. The generated clock frequency is then determined by the value in the `EPPIx_CLKDIV` register.

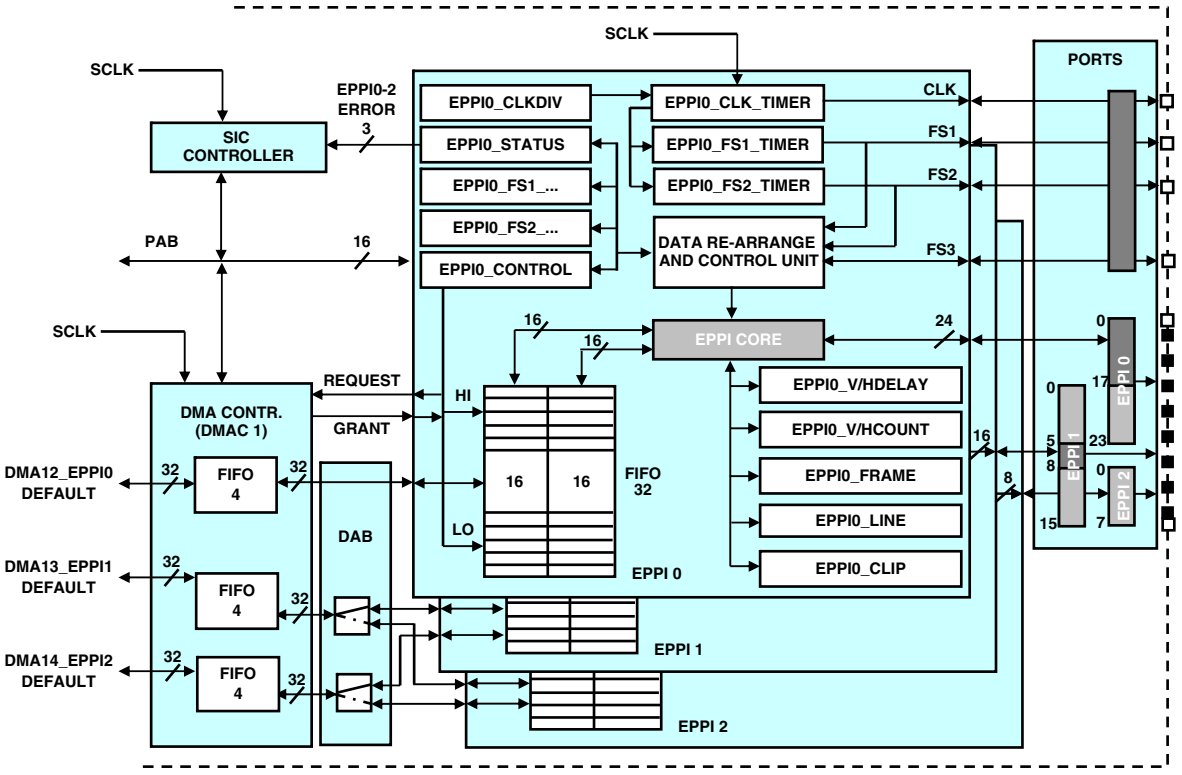


Figure 15-1. EPPI Block Diagram

Description of Operation

The following sections provide descriptions of EPPI operations.

Table 15-1. Operating Modes and Generic EPPI Operation

		How to configure	Useful for	How to configure in ITU R 656 TX Mode
ITU-R BT.656 RX	Entire field	DIR = 0 XFR_TYPE = b#01		
	Active video	DIR = 0 XFR_TYPE = b#00		
	Blanking only	DIR = 0 XFR_TYPE = b#10		
GP 0 FS	TX	DIR = 1 XFR_TYPE = b#11 FS_CFG = b#00	Applications where periodic frame syncs are not used to frame the data	BLANKGEN = 1 DLEN = (b#000, b#001 or b#100)
	RX	DIR = 0 XFR_TYPE = b#11 FS_CFG = b#00		
GP 1 FS	TX	DIR = 1 XFR_TYPE = 11 FS_CFG = 01	Interfacing with ADCs, DACs and other general-purpose devices	BLANKGEN = 1 DLEN = (b#000, b#001 or b#100)
	RX	DIR = 0 XFR_TYPE = b#11 FS_CFG = b#01		

Table 15-1. Operating Modes and Generic EPPI Operation (Cont'd)

		How to configure	Useful for	How to configure in ITU R 656 TX Mode
GP 2 FS	TX	DIR = 1 XFR_TYPE = b#11 FS_CFG = b#10	Video applications that use two hardware synchronization signals, HSYNC and VSYNC	BLANKGEN = 1 DLEN = (b#000, b#001 or b#100)
	RX	DIR = 0 XFR_TYPE = b#11 FS_CFG = b#10		
GP 3 FS	TX	DIR = 1 XFR_TYPE = b#11 FS_CFG = b#11	Video applications that use three hardware sync signals, HSYNC, VSYNC, and FIELD	BLANKGEN = 1 DLEN = (b#000, b#001 or b#100)
	RX	DIR = 0 XFR_TYPE = b#11 FS_CFG = b#11		

EPPI Reset

On a hardware reset, the entire EPPI is reset. All MMRs return to their default values. EPPI interrupt and DMA requests go inactive. Internally generated EPPIx_CLK and frame syncs are aborted.

In software, the EPPI can be reset and re-configured by writing 0 to the EPPIx_EN bit in the EPPIx_CONTROL register. On disabling the EPPI in this manner, only EPPIx_STATUS is cleared to its reset value. EPPI interrupt and DMA requests go inactive, and internally generated clock and frame syncs are aborted.

Clock Gating

In ITU-R BT.656 and GP 0/1/2 FS modes, EPPIx_FS3 becomes a clock-gating input. This is valid for both internally and externally sourced EPPIx_CLK, in both RX and TX modes. This clock gating signal must be synchronous with EPPIx_CLK and must be driven by the external device on the rising edge of EPPIx_CLK. Its function is to hold the sync and data lines in their current state until EPPIx_FS3 is driven low. There are no additional latency cycles upon coming out of clock gating mode.

'If clock gating is not required, the EPPIx_FS3 pin must either be tied to ground, or configured to operate as another of its multiplexed functions.

In GP 2 FS transmit mode with internally generated frame syncs, EPPIx_FS3 functions as a data enable (DEN) pin. Refer to the DEN functionality in the section "GP 2 FS Mode" on page 15-24" for more details on this functionality.

Frame Sync Polarity & Sampling Edge

The POLS and POLC bits provide a mechanism to select the active level of the frame syncs and the sampling/driving edge of the EPPI clock, respectively. This allows the EPPI 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; in these cases, the POLS and POLC bits simply add increased flexibility.

Table 15-2. Different Settings for POLS

	Frame Sync 2	Frame Sync 1
POLS = b#00	Active high/ starts out low	Active high/ starts out low
POLS = b#01	Active high/ starts out low	Active low/ starts out high
POLS = b#10	Active low/ starts out high	Active high/ starts out low
POLS = b#11	Active low/ starts out high	Active low/ starts out high

Table 15-3. Different Settings for POLC

	RX		TX	
	Sample Data	Sample/drive syncs	Drive Data	Sample/drive syncs
POLC = b#00	Falling edge	Falling edge	Rising edge	Rising edge
POLC = b#01	Falling edge	Rising edge	Rising edge	Falling edge
POLC = b#10	Rising edge	Falling edge	Falling edge	Rising edge
POLC = b#11	Rising edge	Rising edge	Falling edge	Falling edge

`EPPIx_FS3` is always active high and starts out as low. In all modes other than GP 3 FS mode, it is used as a clock-gating input, with the exception of when it is configured as a “Data Enable” output in GP 2 FS mode.

Interrupts

The EPPI generates an interrupt to the System Interrupt Controller under the following conditions:

- FIFO Overflow
- FIFO Underflow
- Line Track Overflow
- Line Track Underflow
- Frame Track Overflow
- Frame Track Underflow
- Preamble Error not corrected in ITU-R 656 receive modes

The interrupt remains high until software clears the particular interrupt in the EPPIx_STATUS register.

There is only one interrupt line from each EPPI. An EPPI will therefore internally OR all the above interrupts and send a single interrupt to the core. The EPPIx_STATUS register must then be read to find out which error occurred.

Functional Description

The following sections describe the function of the EPPI.

ITU-R 656 Modes

The EPPI supports three input modes and one output mode for ITU-R 656-framed data. These modes are described in this section.

ITU-R 656 Background

In ITU-656 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.

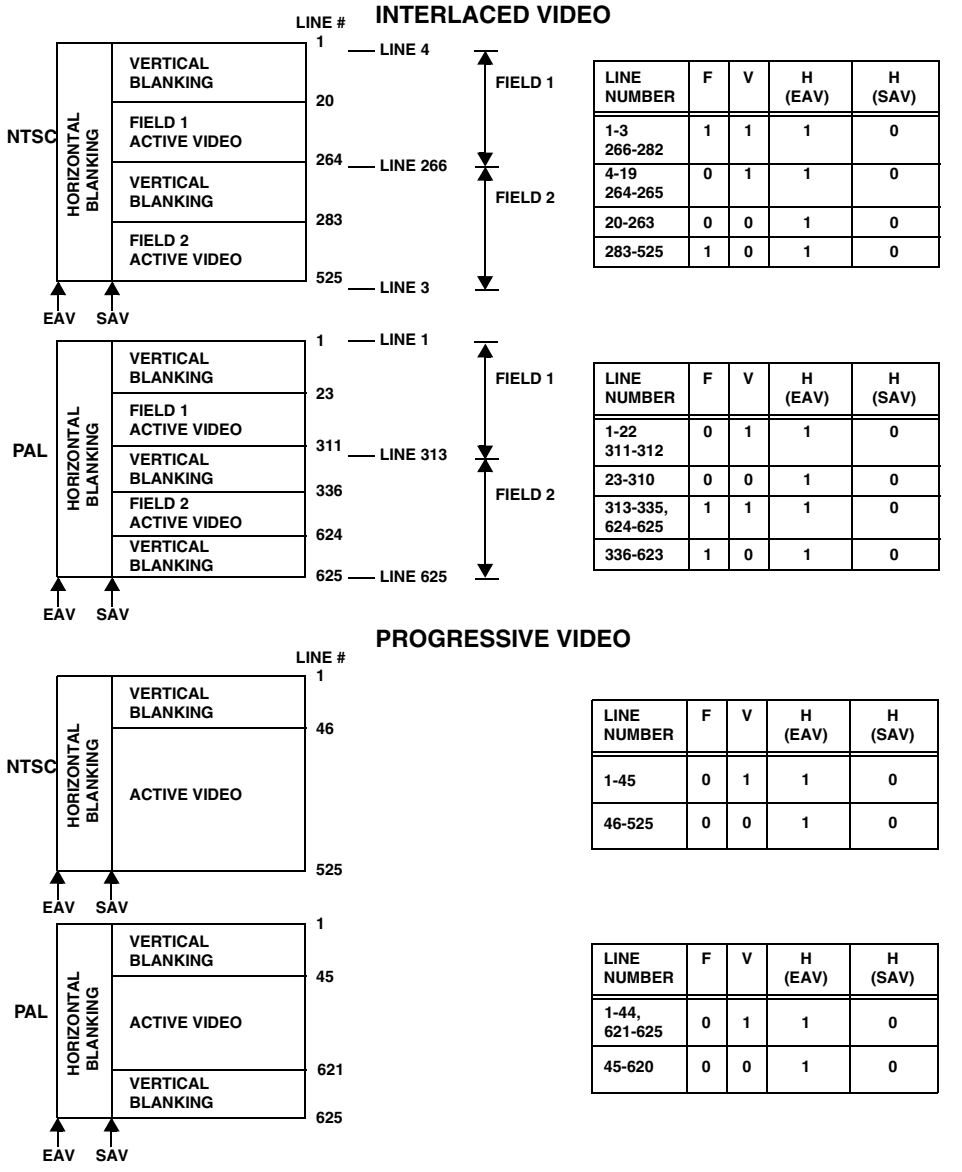
The letter H is used to distinguish between the start of active video (SAV) and end of active video (EAV) signals, which indicate the beginning and end of active video data in each line. SAV occurs on a 1-to-0 transition of H, and EAV occurs on a 0-to-1 transition of H. The space between EAV and SAV is filled with horizontal blanking data. Therefore H = 1 during the horizontal blanking portion of the data stream and H = 0 during the active video portion of the data stream.

The letter V is used to denote the vertical blanking portion of the data stream. A transition in V can occur only in the EAV sequence. When V = 1, the data stream contains vertical blanking data and when V = 0, the data stream contains active video data.

The letter F is used to distinguish Field 1 from Field 2. Interlaced video has two fields in a frame of data. It requires each field to be handled uniquely, and alternate rows of each field combined to create the actual video image.

For interlaced video, F = 0 represents Field 1 and F = 1 represents Field 2. Progressive video makes no distinction between Field 1 and Field 2, and F is always 0 for progressive video.

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 15-3 and Figure 15-2 on page 15-12 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 occurs 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. An “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.



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

LINE NUMBER	F	V	H (EAV)	H (SAV)
1-45	0	1	1	0
46-525	0	0	1	0

LINE NUMBER	F	V	H (EAV)	H (SAV)
1-44, 621-625	0	1	1	0
45-620	0	0	1	0

Figure 15-2. Typical Video Frame Partitioning for NTSC/PAL Systems in Interlaced and Progressive ITU-R BT.656 Systems

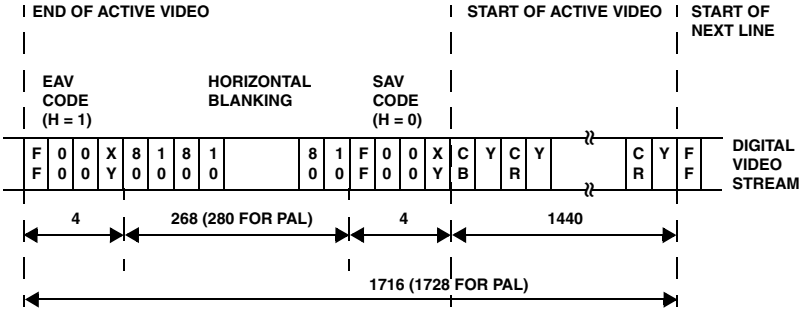


Figure 15-3. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems

The SAV and EAV codes are shown in more detail in Table 15-4. Note there is a defined preamble of three data elements (for example, in the case of 8-bit video: 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 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).

Table 15-4. Control 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

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

P3-P0 are protection bits and enable one- and two-bit errors to be detected, and one-bit errors to be corrected, at the receiver. The EPPI does this correction if it detects one-bit errors in F, V or H. Errors in the protection bits themselves are detected but not corrected.

The EPPIx_STATUS register contains two bits, ERR_DET and ERR_NCOR, used to report the statuses of Error Detected and Error Not Corrected, respectively.

The ERR_DET bit is set whenever an error is detected in the status word. However, this bit does not generate an interrupt. The ERR_NCOR bit is set when more than a 1-bit error is detected in the status word. An interrupt is generated when the ERR_NCOR bit is set. It can be cleared by clearing the ERR_NCOR and ERR_DET bits in the EPPIx_STATUS register. Both bits are sticky and W1C.

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 EPPI 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 are used to delimit fields and frames.

ITU-R 656 Input Modes

Figure 15-4 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.

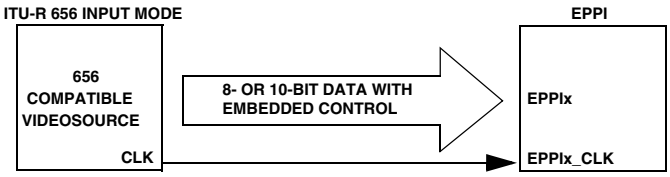


Figure 15-4. ITU-R 656 Input Modes

There are three sub-modes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 15-5 shows these three sub-modes.

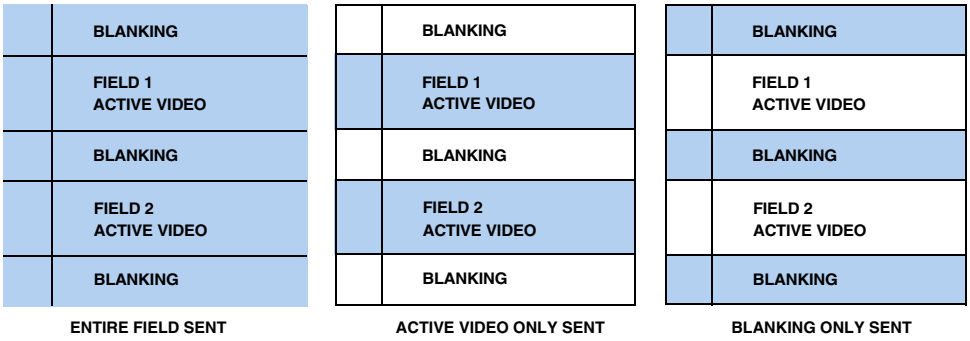


Figure 15-5. ITU-R 656 Input Sub-modes

Entire Field

In this mode, the entire incoming bit stream is read in through the EPPI. 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 for interlaced video, or V = 0 assignment for progressive video.

Active Video

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The EPPI 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 EPPI. After synchronizing to the start of Field 1, the EPPI ignores incoming samples until it sees an SAV.

In this mode, the user must specify the number of total (active plus vertical blanking) lines per frame in the EPPIx_FRAME MMR, and the number of total (active plus horizontal blanking plus 8) samples per line in the EPPIx_LINE 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 EPPI 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.

The VBI is split into two regions within each field. From the EPPI'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.

In this mode, the user must specify the number of total (active plus vertical blanking) lines per frame in the EPPIx_FRAME MMR, and the number of total (active plus horizontal blanking plus 8) samples per line in the EPPIx_LINE MMR.

ITU-R 656 Output in GP Transmit Modes

In GP transmit mode, the EPPI provides the functionality to frame an ITU-R 656 output stream with the proper preambles and blanking intervals. This is done by setting the BLANKGEN bit in the EPPIx_CONTROL register. The EPPI then only needs to fetch active data from memory through the DMA channel, thus saving DMA bandwidth. The EPPIx_AVPL, EPPIx_LVB, EPPIx_LAVF and EPPIx_HBL registers (shown in Figure 15-7) need to be programmed correctly in order for the EPPI to internally generate and embed the proper preamble, status word (EAV and SAV sequences) and blanking data along with the active video from memory. The EPPI can also drive out the frame syncs based on the FS_CFG setting.

Figure 15-6 shows the bit stream format in 16-bit transmit modes with blanking generation (BLANKGEN enabled). Each 16-bit data sample consists of 8-bit Luma (Y) and 8-bit Chroma (Cr or Cb) components.

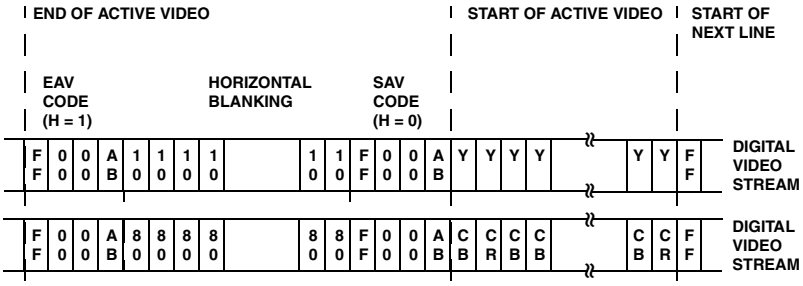


Figure 15-6. 16-Bit Transmit with Internal Blanking Generation

Figure 15-7 shows the data transmitted by the EPPI in this mode. After the EPPI is enabled and if the EPPI FIFO is not empty, the transmission starts by sending out a EAV sequence for a vertical blanking line. For interlaced video, F starts at 1. For progressive video, F is always 0.

Note that the internal blanking generation functionality is valid only when the data length is 8, 10, or 16 bits and when the EPPI is in GP transmit modes. BLANKGEN generates preambles even in GP 2FS mode.

The ITU-R 656 Output mode's internal blanking generation functionality can also be bypassed (for instance, if it is desired to send ancillary data in the blanking interval) by clearing the BLANKGEN bit in the EPPIx_CONTROL register. BLANKGEN generates preambles even in GP 2FS mode.

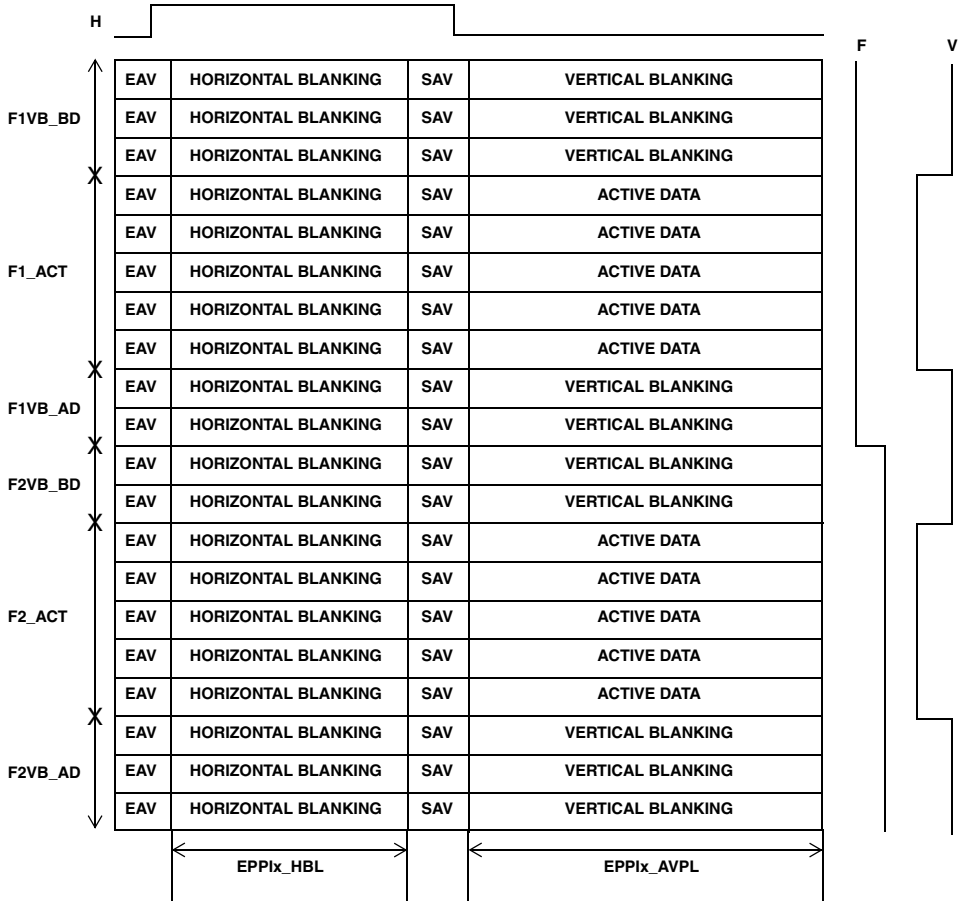


Figure 15-7. Internally Generated Blanking and Preamble Sequence with F, V, and H Signals

Frame Synchronization in ITU-R 656 Modes

For interlaced video, start of frame synchronization happens when a high-to-low transition is detected in F, the field indicator. For progressive video, start of frame synchronization happens when a high-to-low transition is detected in V, the vertical blanking indicator. These transitions in F and V can occur only in the EAV sequence. A start of line is detected on a low-to-high transition in H, the horizontal blanking indicator, and this happens in the EAV sequence as well.

For interlaced video, start of frame corresponds to the start of field 1. Consequently, up to two fields might be ignored (for example, if field 1 just started before the EPPI-to-camera channel was established) before data is received into the EPPI. For progressive video, start of frame corresponds to the start of active video.

Because all H and V signaling is embedded in the data stream in ITU-R 656 modes, the `EPPIx_COUNT` registers are ignored. However, the `EPPIx_FRAME` register is still used in order to check for synchronization errors. Therefore, this MMR must be programmed with the number of lines expected in each frame of video, and the EPPI will keep 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 in the case of interlaced video and transition from V = 1 to V = 0 in the case of progressive video).

At this time, the actual number of lines processed is compared against the value in `EPPIx_FRAME`. If there is a mismatch, a frame track error is asserted in the `EPPIx_STATUS` register. For instance, if an SAV transition was 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 `EPPIx_STATUS` register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose EPPI Modes

The general-purpose (GP) EPPI modes are intended to suit a wide variety of data capture and transmission applications. Each EPPI has three bidirectional frame sync pins. Frame syncs can be generated internally by the EPPI, or by an external device communicating with the EPPI.

GP modes can be distinguished based on the number of frame syncs used. The EPPI supports the following GP modes:

- GP 0 FS mode
- GP 1 FS mode
- GP 2 FS mode
- GP 3 FS mode

GP 0 FS RX mode may be triggered internally or externally. GP 0 FS TX mode is always internally triggered.

All the GP modes, except 0 FS modes, support horizontal windowing. GP modes with 2 and 3 frame syncs also support vertical windowing.

For GP TX modes with internal clock or internal frame syncs, the EPPI will start generating the clock or frame syncs only when the EPPI FIFO has become full for the first time. For GP 0 FS TX mode, the EPPI will only start the transmission when the EPPI FIFO has become full for the first time.

GP 0 FS Mode

GP 0 FS mode is useful for applications where periodic frame syncs are not used to frame the data.

The EPPI can be configured in GP 0 FS mode by setting `XFR_TYPE` = b#11 and `FS_CFG` = b#00 in the `EPPIx_CONTROL` register.

GP 0 FS receive mode is further divided into two sub-modes: internal trigger (FLD_SEL = 0) and external trigger (FLD_SEL = 1), based on how the data transmission/reception is to be initiated. GP 0 FS transmit mode is always internally triggered. All subsequent data manipulation is handled through DMA.

After initial trigger, the EPPI receives/transmits data samples on every clock cycle. However, if SKIPEN is set for receive mode, the EPPI receives only alternate data samples.

The EPPIx_LINE, EPPIx_FRAME, EPPIx_HCOUNT, EPPIx_HDELAY, EPPIx_VCOUNT and EPPIx_VDELAY registers are not valid for GP 0 FS mode. Therefore windowing is not possible in this mode. Also, line and frame track errors are not applicable in this mode.

Frame Synchronization in GP 0 FS External Trigger Mode

When the EPPI is programmed in External Trigger mode, the EPPI will not generate EPPIx_FS1 and a trigger must be provided by the external device. The EPPI starts receiving the data as soon as an EPPIx_FS1 assertion is detected. After that, all subsequent data manipulation is handled by way of DMA and any activity on EPPIx_FS1 is ignored.

Frame Synchronization in GP 0 FS Internal Trigger Mode

When the EPPI is programmed in internal trigger mode, the EPPI starts receiving/transmitting data as soon as the EPPI clock is enabled and synchronized.

Note that GP 0 FS transmit mode is always internally triggered. The EPPI starts transmitting valid data when the EPPI FIFO becomes full and the EPPI clock is enabled. Care should be taken that the clock is enabled only after the EPPI FIFO becomes full.

There may be up to two cycles latency before valid data is received or transmitted.

GP 1 FS Mode

GP 1 FS mode is useful for interfacing the EPPI with analog-to-digital converters (ADCs), digital-to-analog converters (DACs) and other general- purpose devices. This mode works for both transmit and receive.

The EPPI can be configured in GP 1 FS mode by setting XFR_TYPE = b#11 and FS_CFG = b#01 in the EPPIx_CONTROL register. The frame sync may be provided by an external device or it can be sourced by the EPPI itself.

The EPPI windowing registers must be carefully programmed in GP 1 FS mode such that:

- The EPPIx_LINE register contains the number of clock cycles expected between two assertions of EPPIx_FS1. This is used to keep track of Line Track errors. It must be programmed before the EPPIx_HCOUNT register.
- The EPPIx_HDELAY register contains the number of clock cycles to wait after the assertion of EPPIx_FS1, for example, start of frame.
- The EPPIx_HCOUNT register contains the number of data samples to receive or transmit for each frame.

The EPPIx_FRAME, EPPIx_VDELAY and EPPIx_VCOUNT registers have no effect in GP 1 FS mode. As a result, frame track errors and vertical windowing are not possible in this mode.

GP 2 FS Mode

GP 2 FS mode is useful for video applications that use two hardware synchronization signals, HSYNC and VSYNC. The HSYNC can be connected to EPPIx_FS1 and VSYNC can be connected to EPPIx_FS2.

The EPPI can be configured in GP 2 FS mode by setting XFR_TYPE = b#11 and FS_CFG = b#10 in the EPPIx_CONTROL register. The frame syncs may be provided by an external device or they can be sourced by the EPPI itself.

The EPPI windowing registers must be programmed for GP2 FS mode in the sequence listed below.

The EPPI windowing registers must be carefully programmed in GP 2 FS mode such that:

- The EPPIx_FRAME register contains the number of expected lines per frame. It should be equal to the number of EPPIx_FS1 assertions expected between start of frame syncs and is used to keep track of frame track errors. It must be programmed before the EPPIx_VCOUNT register.
- The EPPIx_LINE register contains the number of clock cycles expected between two assertions of EPPIx_FS1. This is used to keep track of line track errors. It must be programmed before the EPPIx_HCOUNT register.
- The EPPIx_HDELAY register contains the number of clock cycles to wait after the assertion of EPPIx_FS1, for example, start of line.
- The EPPIx_HCOUNT register contains the number of data samples to receive or transmit for each line.
- The EPPIx_VDELAY register contains the number of lines to wait after the start of frame is detected.
- The EPPIx_VCOUNT register contains the number of lines to receive or transmit.

DEN functionality in GP 2 FS Transmit Mode

When EPPI is configured in GP 2 FS TX mode and when the EPPI is configured for internal frame sync generation, the EPPIx_FS3 pin functions as a data enable (DEN) pin. The functionality of the DEN pin is described in the following two cases:

Case 1 - When blanking generation (BLANKGEN) is enabled and the EPPI data length (DLEN) is configured for 8-, 10-, or 16-bit transfers:

The EPPIx_FS3 pin will assert during the “active data” regions, aligned with EPPIx_CLK according to the clock polarity (POLC) settings. The frame sync polarity (POLS) setting does not apply here -- EPPIx_FS3 will always be active high in this mode.

Case 2 - When blanking generation (BLANKGEN) is disabled or it is enabled but the EPPI data length (DLEN) is configured for a transfer size different from 8-, 10-, or 16-bits:

The EPPIx_FS3 pin will assert at the start of the active data region on each line, aligned with EPPIx_CLK according to the clock polarity (POLC) settings. The frame sync polarity (POLS) setting does not apply here -- EPPIx_FS3 will always be active high in this mode. Once asserted, EPPIx_FS3 will stay asserted for EPPIx_HCOUNT number of clock cycles per line, and then it will deassert. This behavior on each line will continue for the total number of lines programmed in EPPIx_VCOUNT per frame, and repeat at the start of subsequent video frames.

In case 2, if transmission of valid data is held off due to delays programmed in the EPPIx_HDELAY and/or EPPIx_VDELAY registers, the assertion of EPPIx_FS3 will also be held off accordingly, on a per-line and/or per-frame basis.

GP 3 FS Mode

GP 3 FS mode is useful for video applications that use three hardware synchronization signals, HSYNC, VSYNC, and FIELD. The HSYNC can be connected to EPPIx_FS1, VSYNC can be connected to EPPIx_FS2, and FIELD can be connected to EPPIx_FS3.

The EPPI can be configured in GP 3 FS mode by setting XFR_TYPE = b#11 and FS_CFG = b#11 in the EPPIx_CONTROL register. The frame syncs may be provided by an external device or they can be sourced by the EPPI itself.

GP 3 FS mode is very much similar in operation to GP 2 FS mode, except that the Start of Frame synchronization in GP 3 FS mode also takes into account EPPIx_FS3. All the windowing register (EPPIx_FRAME, EPPIx_LINE, EPPIx_HDELAY, EPPIx_HCOUNT, EPPIx_VDELAY and EPPIx_VCOUNT) settings, as well as data reception/transmission and error generation are the same as for GP 2 FS mode. In addition, for GP 3 FS mode with internal frame syncs, the FLD_SEL bit setting specifies the condition under which the transfer should begin.

EPPI Data Path Options

The EPPI data path options are described in this section.

EPPI Data Lengths

EPPI data lengths are configured by setting the DLEN bits in the EPPIx_CONTROL register.

EPPI0 supports data lengths of 8, 10, 12, 14, 16, 18 or 24 bits.

EPPI1 supports data lengths of 8, 10, 12, 14, or 16 bits.

EPPI2 supports only 8-bit data.

The EPPI1 data pins are multiplexed with EPPI2 data pins and some EPPI0 data pins (This is shown visually in Figure 15-8). For more information see the PORT_MUX register description in Chapter 9, "General-Purpose Ports".

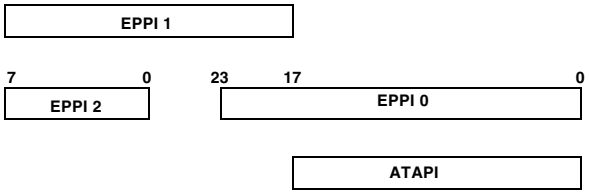


Figure 15-8. EPPI Pin Muxing

EPPI DMA Channels

Each EPPI has a 32-bit DMA channel connected to it. In addition, if EPPI2 is unused, EPPI1 may use that DMA channel as an additional DMA channel. Similarly, if EPPI1 is unused, EPPI2 may use that DMA channel as an additional DMA channel. However, this second channel is enabled only when the DMACFG bit is set in the EPPIx_CONTROL register.

Data Packing For Receive Modes

For receive modes, if PACKEN is set in EPPIx_CONTROL, the DMA is a 32-bit DMA and the EPPI packs the incoming data into 32-bit words based on the DLEN and SWAPEN bit settings. When SWAPEN = 0, the EPPI puts the first data in the least significant bits and when SWAPEN = 1, the EPPI puts the first data in the most significant bits. Following are the packing options based on the DLEN bits:

- When DLEN = 8, four 8-bit words can be packed into one 32-bit word.
- When DLEN = 16, two 16-bit words can be packed into one 32-bit word.
- For DLEN values that are more than 8 bits but less than 16 bits, two such words are either sign-extended or zero-filled, and packed into one 32-bit word.
- When DLEN = 18, the EPPI sign-extends or zero-fills the 18-bit data to 24 bits and packs four 24-bit words into three 32-bit words.
- When DLEN = 24, the EPPI packs four 24-bit words into three 32-bit words.

  When PACKEN is cleared in the EPPIx_CONTROL register, the EPPI receives the incoming data and sends it on the DAB bus as-is. If DLEN is less than or equal to 16 bits, the DMA is a 16-bit DMA; otherwise it is a 32-bit DMA. Examples of data packing are provided in "Data Transfer Examples" on page 15-33.

Data Unpacking For Transmit Modes

For transmit modes, if PACKEN is set in EPPIx_CONTROL, the DMA is a 32-bit DMA and the EPPI unpacks the 32-bit word according to the DLEN and SWAPEN bit settings.

If SWAPEN = 1, the EPPI transmits the most significant bits as the first data, and if SWAPEN = 0, the EPPI transmits the least significant bits as the first data. Following are the various unpacking modes, based on the DLEN bits:

- When DLEN = 8, the EPPI transmits one 32-bit word from memory as four 8-bit data words.
- For DLEN values greater than 8 bits but less than or equal to 16 bits, the EPPI transmits one 32-bit word from memory as two data words.
- When DLEN = 18 or DLEN = 24, the EPPI transmits three 32-bit words from memory as four data words. Examples of data unpacking are provided in "Data Transfer Examples" on page 15-33.

Sign-Extension and Zero-Filling

For DLEN equal to 10, 12 or 14, data is zero-filled or sign-extended to 16 bits.

For DLEN equal to 18 bits, data is zero-filled or sign-extended to 24 bits if packing is enabled, and zero-filled or sign-extended to 32 bits if packing is disabled.

For DLEN equal to 24 bits, data is zero-filled or sign-extended to 32 bits if packing is disabled.

For DLEN equal to 8 bits, data is zero-filled or sign-extended to 16 bits if packing is disabled.

If the SIGN_EXT bit in the EPPIx_CONTROL register is set (SIGN_EXT = 1), then the data is sign-extended, otherwise it is zero-filled.

Split Receive Modes

The EPPIx_CONTROL register has three control bits for Split receive modes. These are SPLT_EVEN_ODD, SUBSPLT_ODD and DMACFG. PACKEN is not valid in Split modes.

If SPLT_EVEN_ODD is set, the EPPI splits the incoming data stream into two sub-streams, an even stream and an odd stream, and packs them separately.

SUBSPLT_ODD is valid only if SPLT_EVEN_ODD is set. If SUBSPLT_ODD is set, the EPPI sub-splits the odd sub-stream, and packs them separately.

DMACFG is also valid only if SPLT_EVEN_ODD is set. If DMACFG is set, the EPPI uses two DMA channels and if DMACFG is cleared, the EPPI uses only one DMA channel.

Split mode can only be used on EPPI1 or EPPI2. Examples are provided in "Data Transfer Examples" on page 15-33.

Split Transmit Modes

The EPPIx_CONTROL register has three control bits for Split transmit modes. These are SPLT_EVEN_ODD, SUBSPLT_ODD and DMACFG. The DMA is always a 32-bit DMA. PACKEN is not valid in Split modes.

If SPLT_EVEN_ODD is set, the EPPI receives the Luma ($Y_3Y_2Y_1Y_0$) and interleaved Chroma ($Cr_1Cb_1Cr_0Cb_0$) data as 32 bits from the DMA channel and interleaves them to form a 4:2:2 YCrCb data stream to be transmitted out.

SUBSPLT_ODD is valid only if SPLT_EVEN_ODD is set. If SUBSPLT_ODD is set, the EPPI receives the Luma ($Y_3Y_2Y_1Y_0$) and de-interleaved Chroma ($Cb_3Cb_2Cb_1Cb_0$ and $Cr_3Cr_2Cr_1Cr_0$) and interleaves them to form a 4:2:2 YCrCb data stream to be transmitted out.

DMACFG is also valid only if SPLT_EVEN_ODD is set. If DMACFG is set, the EPPI uses two DMA channels and if DMACFG is cleared, the EPPI uses only one DMA channel.

Split mode can only be used on EPPI1 or EPPI2. Examples are provided in "Data Transfer Examples" on page 15-33.

RGB Data Formats

For transmit modes, the EPPI can convert RGB888 data in memory to RGB666 at the output if the RGB_FMT_EN bit is set in the EPPIx_CONTROL register and if DLEN is equal to 18 bits. Similarly, the EPPI can convert RGB888 data in memory to RGB565 at the output if the RGB_FMT_EN bit is set in the EPPIx_CONTROL register and if DLEN is equal to 16 bits.

This conversion is done as follows: if PACKEN = 1, the EPPI first unpacks, according to the SWAPEN settings, the three 32-bit data words from the DMA into four 24-bit data words to be transmitted out as described earlier. If PACKEN = 0, then the EPPI takes the lower 24 bits of the 32-bit DMA as the data to be transmitted. Then the EPPI truncates this 24-bit data word to the required data width by removing the lower 2 bits of G and the lower 2 or 3 bits of R and B.

SPLT_EVEN_ODD and RGB_FMT_EN should never be set simultaneously.

Programmed Clipping and Thresholding of Data Values

The EPPI supports clipping and thresholding of data values for transmit modes. This feature is valid only when the data length is 8 or 16 bits.

The EPPIx_CLIP register is used to define the lower and upper limits for the Luma and Chroma components.

The bit definitions of this register are shown in Table 15-5.

Table 15-5. EPPIx_CLIP Memory Mapped Register

Bits	Name	Description
7:0	LOW_ODD	Lower Limit for Odd Bytes (Chroma)
15:8	HIGH_ODD	Upper Limit for Odd Bytes (Chroma)
23:16	LOW_EVEN	Lower Limit for Even Bytes (Luma)
31:24	HIGH_EVEN	Upper Limit for Even Bytes (Luma)

For the 4:2:2 YCrCb color space, Luma and Chroma typically have different lower and upper thresholds, which is why separate thresholds may be required for even and odd data samples. In the case of monochrome (Y only) or some non-video clipping applications, LOW_ODD should be the same as LOW_EVEN, and HIGH_ODD should be the same as HIGH_EVEN.

For 16-bit data lengths, the EPPI will separate each word into upper and lower bytes, and will consider the lower bytes as odd bytes and the upper bytes as even bytes during clipping.

Data Transfer Examples

The following sections provide EPPI data transfer examples.

8-Bit Receive Mode

For 8-bit non-split receive mode, the EPPI will pack four bytes of incoming data into one 32-bit word, if PACKEN = 1 in the EPPIx_CONTROL register. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_EO bits. The first incoming data can be placed either in the least significant bit positions or in the most significant bit positions, based on the SWAPEN bit setting.

Table 15-6 shows an 8-bit receive mode example when PACKEN = 1.

Table 15-6. Data Received in 8-Bit Receive Mode with Packing Enabled

Pin Data (8 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 0 SIGN_EXT = X	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 1 SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 0 SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 0 SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 1 SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 1 SIGN_EXT = X
0x11						
0x22						
0x33						
0x44	0x4433 2211	0x1122 3344				
0x55						
0x66						
0x77			0x7755 3311		0x1133 5577	
0x88	0x8877 6655	0x5566 7788		0x8866 4422		0x2244 6688
0x99						
0xAA						
0xBB						
0xCC	0xCCBB AA99	0x99AA BBCC				
0xDD						
0xEE						
0xFF			0xFFDD BB99		0x99BB DDFF	
0x00	0x00FF EEDD	0xDDE EFF00		0x00EE CCAA		0xAACC EE00

If PACKEN = 0, the DMA is a 16-bit DMA and the EPPI either sign-extends or zero-fills the bytes of incoming data into a 16-bit word. SWAPEN has no effect if PACKEN = 0.

Table 15-7 shows an 8-bit receive mode example when PACKEN = 0:

Table 15-7. Data Received in 8-Bit Receive Mode with Packing Disabled

Pin Data (8 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 1	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = 1
0x44	0x0044	0x0044	0x0044	
0x55	0x0055	0x0055		0x0055
0x66	0x0066	0x0066	0x0066	
0x77	0x0077	0x0077		0x0077
0x88	0x0088	0xFF88	0x0088	
0x99	0x0099	0xFF99		0xFF99
0xAA	0x00AA	0xFFAA	0x00AA	
0xBB	0x00BB	0xFFBB		0xFFBB

10/12/14-Bit Receive Modes

For 10-, 12-, or 14-bit non-split receive modes, the EPPI will first either zero-fill or sign-extend the incoming data (depending on the SIGN_EXT bit) into a 16-bit word. If PACKEN = 1, the EPPI will then pack two of these words into one 32-bit word. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_EO bits. The first incoming data can be placed either in the least significant bit positions or in the most significant bit positions, based on the SWAPEN bit setting.

Table 15-8 shows a 10-Bit receive mode example when PACKEN = 1 and SIGN_EXT = 1:

Table 15-8. Data Received in 10-Bit Receive Mode with Sign Extension, with Packing Enabled

Pin Data (10 bits)	MSB	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 0 SIGN_EXT=1	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 1 SIGN_EXT=1	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 0 SIGN_EXT=1	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN= 0 SIGN_EXT=1	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 1 SIGN_EXT=1	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 1 SIGN_EXT=1
0x111	0						
0x222	1	0xFE22 0111	0x0111 FE22				
0x333	1			0xFF33 0111		0x0011 FF33	
0x044	0	0x0044 FF33	0xff33 0044		0x0044 FE22		0xFE22 0044
0x155	0						
0x266	1	0xFE66 0155	0x0155 FE66				
0x377	1			0xFF77 0155		0x0155 FF77	
0x088	0	0x0088 FF77	0xFF77 0088		0x0088 FE66		0xFE66 0088

Table 15-9 Shows a 10-Bit Receive Mode Example when PACKEN = 1 and SIGN_EXT = 0:

Table 15-9. Data Received in 10-Bit Receive Mode, with Zero-Fill, with Packing Enabled

Pin Data (10 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 0 SIGN_EXT=0	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 1 SIGN_EXT=0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 0 SIGN_EXT=0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 0 SIGN_EXT=0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 1 SIGN_EXT=0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 1 SIGN_EXT=0
0x111						
0x222	0x0222 0111	0x0111 0222				
0x333			0x0333 0111		0x0011 0333	
0x044	0x0044 0333	0x0333 0044		0x0044 0222		0x0222 0044
0x155						
0x266	0x0266 0155	0x0155 0266				
0x377			0x0377 0155		0x0155 0377	
0x088	0x0088 0377	0x0377 0088		0x0088 0266		0x0266 0088

Table 15-10 shows a 10-bit receive mode example when PACKEN = 0:

Table 15-10. Data Received in 10-bit Receive Mode with Packing Disabled

Pin Data (10 bits)	MSB	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 1	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = 1	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = 0
0x111	0	0x0111	0x0111	0x0111	
0x222	1	0xFE22	0x0222		0x0222
0x333	1	0xFF33	0x0333	0xFF33	
0x044	0	0x0044	0x0444		0x0444
0x155	0	0x0155	0x0155	0x0155	
0x266	1	0xFE66	0x0266		0x0266
0x377	1	0xFF77	0x0377	0xFF77	
0x088	0	0x0088	0x0088		0x088

16-Bit Receive Mode

For 16-bit non-split receive mode, the EPPI will pack two 16-bit incoming data into one 32-bit word, if PACKEN = 1. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_EO bits. The first incoming data can be placed either in the least significant bit positions or in the most significant bit positions, based on the SWAPEN bit setting.

Table 15-11 shows a 16-bit receive mode example when PACKEN = 1.

Table 15-11. Table 6: Data Received in 16-Bit Receive Mode with Packing Enabled

Pin Data (16 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 0 SIGN_EXT=X	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = 1 SIGN_EXT=X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 0 SIGN_EXT=X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 0 SIGN_EXT=X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = 1 SIGN_EXT=X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = 1 SIGN_EXT=X
0x1111						
0x2222	0x2222 1111	0x1111 2222				
0x3333			0x3333 1111		0x1111 3333	
0x4444	0x4444 3333	0x3333 4444		0x4444 2222		0x2222 4444
0x5555						
0x6666	0x6666 5555	0x5555 6666				
0x7777			0x7777 5555		0x5555 7777	
0x8888	0x8888 7777	0x7777 8888		0x8888 6666		0x6666 8888

Table 15-12 shows a 16-bit receive mode example when PACKEN = 0:

Table 15-12. Data Received in 16-bit Receive Mode with Packing Disabled

Pin Data (16 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = X
0x1111	0x1111	0x1111	
0x2222	0x2222		0x2222
0x3333	0x3333	0x3333	
0x4444	0x4444		0x4444
0x5555	0x5555	0x5555	
0x6666	0x6666		0x6666
0x7777	0x7777	0x7777	
0x8888	0x8888		0x8888

18-Bit Receive Mode

For 18-bit non-split receive mode, the EPPI will zero-fill or sign-extend the incoming data into a 32-bit word, if PACKEN = 0. If PACKEN = 1, the EPPI will first zero-fill or sign-extend the incoming data to 24 bits, and then pack four such 24-bit data words into three 32-bit words. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_EO bits. The SWAPEN bit has no effect.

Table 15-13 shows an 18-bit receive mode example when PACKEN = 0:

Table 15-13. Data Received in 18-bit Receive Mode with Packing Disabled

Pin Data (18 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = 0
0x0 6666	0x0000 6666	0x0000 6666	
0x1 7777	0x0001 7777		0x0001 7777
0x2 8888	0x0002 8888	0x0002 8888	
0x3 9999	0x0003 9999		0x0003 9999

Table 15-14 shows an 18-bit receive mode example when PACKEN = 1:

Table 15-14. Data Received in 18-bit Receive Mode with Packing Enabled

Pin Data (18 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = 0
0x0 1122			
0x1 3344	0x4400 1122		
0x2 5566	0x5566 0133	0x6600 1122	
0x3 7788	0x0377 8802		0x8801 3344
0x0 99AA		0x99AA 0255	
0x1 BBCC	0xCC00 99AA		0xBBCC 0377
0x2 DDEE	0xDDEE 01BB	0x02DD EE00	
0x3 FF12	0x03FF 122D		0x03FF 1201

24-Bit Receive Mode

For 24-bit non-split receive mode, the EPPI will zero-fill or sign-extend the incoming data into a 32-bit word, if PACKEN = 0. If PACKEN = 1, the EPPI will pack four incoming 24-bit data words into three 32-bit words. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_EO bits. The SWAPEN bit has no effect.

Table 15-15 shows a 24-bit receive mode example when PACKEN = 0:

Table 15-15. Data Received in 24-bit Receive Mode with Packing Disabled

Pin Data (24 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X SIGN_EXT = 0	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X SIGN_EXT = 0
0x6 6666	0x0066 6666	0x0066 6666	
0x7 7777	0x0077 7777		0x0077 7777
0x8 8888	0x0088 8888	0x0088 8888	
0x9 9999	0x0099 9999		0x0099 9999

Table 15-16 shows a 24-bit receive mode example when PACKEN = 1:

Table 15-16. Data Received in 24-bit Receive Mode with Packing Enabled

Pin Data (24 bits)	DMA DATA when SKIP_EN = 0 SKIP_EO = X SWAPEN = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 1 SWAPEN = X	DMA DATA when SKIP_EN = 1 SKIP_EO = 0 SWAPEN = X
0x11 2233			
0x44 5566	0x6611 2233		
0x77 8899	0x8899 4455	0x9911 2233	
0x00 AABB	0x00AA BB77		0xBB44 5566
0xCC DDEE		0xDDEE 7788	
0xFF 1234	0x34CC DDEE		0x1234 00AA
0x56 7890	0x7890 FF12	0x5678 90CC	
0xAB CDEF	0xABCD EF56		0xABCD EFFF

8-Bit Split Receive Mode

For 8-bit split receive mode, PACKEN and SIGN_EXT are not valid. The EPPI always packs four bytes of data into one 32-bit word.

Table 15-17 shows an 8-bit split receive mode example with SWAPEN = 0 and SKIP_EN = 0:

Table 15-17. Data Received in 8-bit Split Receive Mode with SKIP_EN = 0 and SWAPEN = 0

Pin Data (8 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG =0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
V_0						
Y_0						
U_0						
Y_1						
V_1						
Y_2						
U_1		$U_1V_1U_0V_0$	$U_1V_1U_0V_0$			
Y_3	$Y_3Y_2Y_1Y_0$		$Y_3Y_2Y_1Y_0$	$Y_3Y_2Y_1Y_0$		$Y_3Y_2Y_1Y_0$
V_2						
Y_4						
U_2						
Y_5						
V_3					$V_3V_2V_1V_0$	$V_3V_2V_1V_0$
Y_6						
U_3		$U_3V_3U_2V_2$	$U_3V_3U_2V_2$		$U_3U_2U_1U_0$	
Y_7	$Y_7Y_6Y_5Y_4$		$Y_7Y_6Y_5Y_4$	$Y_7Y_6Y_5Y_4$		$Y_7Y_6Y_5Y_4$
V_4						$U_3U_2U_1U_0$

Table 15-18 shows an 8-bit split receive mode example with SWAPEN = 1 and SKIP_EN = 0:

Table 15-18. Data Received in 8-bit Split Receive Mode with SKIP_EN = 0 and SWAPEN = 1

Pin Data (8 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG = 0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
V_0						
Y_0						
U_0						
Y_1						
V_1						
Y_2						
U_1		$V_0U_0V_1U_1$	$V_0U_0V_1U_1$			
Y_3	$Y_0Y_1Y_2Y_3$		$Y_0Y_1Y_2Y_3$	$Y_0Y_1Y_2Y_3$		$Y_0Y_1Y_2Y_3$
V_2						
Y_4						
U_2						
Y_5						
V_3					$V_0V_1V_2V_3$	$V_0V_1V_2V_3$
Y_6						
U_3		$V_2U_2V_3U_3$	$V_2U_2V_3U_3$		$U_0U_1U_2U_3$	

Table 15-18. Data Received in 8-bit Split Receive Mode with SKIP_EN = 0 and SWAPEN = 1 (Cont'd)

Pin Data (8 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG = 0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
Y_7	$Y_4Y_5Y_6Y_7$		$Y_4Y_5Y_6Y_7$	$Y_4Y_5Y_6Y_7$		$Y_4Y_5Y_6Y_7$
V_4						$U_0U_1U_2U_3$

For the case when SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 1 and DMACFG = 0, note that although the second Chroma component ($U_0U_1U_2U_3$ in Table 15-16) sent over the DMA bus is completely packed before the Luma component ($Y_4Y_5Y_6Y_7$ in Table 15-16), it is intentionally held until that previous word is moved out. This is done in order to enable the separation of Luma and Chroma values into individual buffers when using 2D-DMA.

10/12/14/16-Bit Split Receive Mode with SPLT_16 = 0

For 16-bit split receive mode, PACKEN is not valid. The EPPI always packs two 16-bit words into one 32-bit word. For 10-, 12-, or 14-bit split receive modes, the EPPI will first either sign-extend or zero-fill the incoming data into a 16 bit word, and then pack two of these words into one 32-bit word to be sent to the DMA.

Table 15-19 shows a 16-bit split receive mode example with SWAPEN = 0 and SKIP_EN = 0:

Table 15-19. Data received in 16-bit split receive mode with SPLT_16 = 0, SKIP_EN = 0 and SWAPEN = 0

Pin Data (16 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG = 0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
V_0						
Y_0						
U_0		U_0V_0	U_0V_0			
Y_1	Y_1Y_0		Y_1Y_0	Y_1Y_0		Y_1Y_0
V_1					V_1V_0	V_1V_0
Y_2						
U_1		U_1V_1	U_1V_1		U_1U_0	
Y_3	Y_3Y_2		Y_3Y_2	Y_3Y_2		Y_3Y_2
V_2						U_1U_0

Table 15-20 shows an 16-bit split receive mode example with SWAPEN = 1 and SKIP_EN = 0:

Table 15-20. Data received in 16-bit split receive mode with SPLT_16 = 0, SKIP_EN = 0 and SWAPEN = 1

Pin Data (16 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 1 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG = 0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
V_0						
Y_0						
U_0		V_0U_0	V_0U_0			
Y_1	Y_0Y_1		Y_0Y_1	Y_0Y_1		Y_0Y_1
V_1					V_0V_1	V_0V_1
Y_2						
U_1		V_1U_1	V_1U_1		U_0U_1	
Y_3	Y_2Y_3		Y_2Y_3	Y_2Y_3		Y_2Y_3
V_2						U_0U_1

16-Bit Split Receive Mode with SPLT_16 = 1

For 16-bit split receive mode, PACKEN is not valid. The EPPI always packs two 16-bit words into one 32-bit word. The SPLT_16 bit is only valid when DLEN = 16 bits.

Table 15-21 shows a 16-bit split receive mode example with SPLT_16 = 1, SWAPEN = 0 and SKIP_EN = 0:

Table 15-21. Data Received in 16-bit Split Receive Mode with SPLT_16 = 1, SKIP_EN = 0 and SWAPEN = 0

Pin Data (16 bits)	SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X			SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 0 SKIP_EN = 0 SKIP_EO = X		
	DMACFG = 1		DMACFG = 0	DMACFG = 1		DMACFG = 0
	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL	PRIMARY DMA CHANNEL	SECONDARY DMA CHANNEL	PRIMARY DMA CHANNEL
V_0Y_0						
U_0Y_1						
V_1Y_2						
U_1Y_3	$Y_3Y_2Y_1Y_0$	$U_1V_1U_0V_0$	$Y_3Y_2Y_1Y_0$	$Y_3Y_2Y_1Y_0$		$Y_3Y_2Y_1Y_0$
V_2Y_4			$U_1V_1U_0V_0$			
U_2Y_5						
V_3Y_6					$V_3V_2V_1V_0$	$V_3V_2V_1V_0$
U_3Y_7	$Y_7Y_6Y_5Y_4$	$U_3V_3U_2V_2$	$Y_7Y_6Y_5Y_4$	$Y_7Y_6Y_5Y_4$	$U_3U_2U_1U_0$	$Y_7Y_6Y_5Y_4$
V_4Y_8			$U_3V_3U_2V_2$			$U_3U_2U_1U_0$

8-Bit Transmit Mode

For 8-bit non-split transmit mode, if PACKEN = 1, the DMA is a 32-bit DMA and the EPPI will unpack the 32-bit word from memory into four bytes to be transmitted out. The EPPI transmits either the most significant bits or the least significant bits as the first data, depending on the SWAPEN bit setting. If PACKEN = 0, the DMA is a 16-bit DMA and the EPPI transmits the lower 8 bits. SWAPEN has no effect when PACKEN = 0.

Table 15-22 shows an 8-bit transmit mode example when PACKEN = 1:

Table 15-22. Data Sent in 8-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN = 0	Pin Data when SWAPEN = 1
0x1122 3344	0x44	0x11
0x5566 7788	0x33	0x22
	0x22	0x33
	0x11	0x44
	0x88	0x55
	0x77	0x66
	0x66	0x77
	0x55	0x88

Table 15-23 shows a 8-bit transmit mode example when PACKEN = 0:

Table 15-23. Data Sent in 8-bit Transmit Mode with Packing Disabled

DMA Data (16 bits)	Pin Data SWAPEN = X
0x1234	0x34
0x2345	0x45
0x3456	0x56

10/12/14-Bit Transmit Modes

For 10-, 12-, or 14-bit non-split transmit modes, if PACKEN = 1, the DMA is a 32-bit DMA and the EPPI will unpack the 32-bit word from memory into two 16-bit data words, then transmit the required least significant bits from each. The EPPI transmits either the most significant word or the least significant word as the first data, depending on the SWAPEN bit setting. If PACKEN = 0, the DMA is a 16-bit DMA and the EPPI transmits the required least significant bits. SWAPEN has no effect when PACKEN = 0.

Table 15-24 shows a 10-bit transmit mode example when PACKEN = 1:

Table 15-24. Data Sent in 10-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN = 0	Pin Data when SWAPEN = 1
0x1111 2222	0x222	0x111
0x3333 4444	0x111	0x222
	0x044	0x333
	0x333	0x044

Table 15-25 shows a 10-bit transmit mode example when PACKEN = 0:

Table 15-25. Data Sent in 10-bit Transmit Mode with Packing Disabled

DMA Data (16 bits)	Pin Data SWAPEN = X
0x1234	0x234
0x2345	0x345
0x3456	0x056
0x4567	0x167

16-Bit Transmit Mode

For 16-bit non-split transmit mode, if PACKEN = 1, the DMA is a 32-bit DMA and the EPPI will unpack the 32-bit word from memory into two 16-bit data words to be transmitted out. The EPPI transmits either the most significant bits or the least significant bits as the first data, depending on the SWAPEN bit setting. If PACKEN = 0, the DMA is a 16-bit DMA and the EPPI transmits the data as-is. SWAPEN has no effect when PACKEN = 0.

Table 15-26 shows a 16-bit transmit mode example when PACKEN = 1:

Table 15-26. Data Sent in 16-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN = 0	Pin Data when SWAPEN = 1
0x1111 2222	0x2222	0x1111
0x3333 4444	0x1111	0x2222
	0x4444	0x3333
	0x3333	0x4444

Table 15-27 shows a 16-bit transmit mode example when PACKEN = 0:

Table 15-27. Data Sent in 16-bit Transmit Mode with Packing Disabled

DMA Data (16 bits)	Pin Data SWAPEN = X
0x1234	0x1234
0x2345	0x2345
0x3456	0x3456

18-Bit Transmit Mode

For 18-bit transmit mode, if PACKEN = 1, the DMA is a 32-bit DMA and the EPPI will unpack the 32-bit word from memory.

Table 15-28 shows a 18-bit transmit mode example when PACKEN = 1. Note that when RGB_FMT_EN is set, the least significant bits of R, G, and B are dropped.

Table 15-28. Data Sent, 18-bit Transmit Mode with Packing Enabled

DMA Data	Pin Data (18-bits)	
	RGB_FMT_EN = 0	RGB_FMT_EN = 1
0x0123 4567	0x3 4567	0x0 8459
0x89AB CDEF	0x1 EF01	0x3 3EC0
0x0123 4567	0x3 89AB	0x1 98AA
	0x1 2345	0x0 0211

Table 15-29 shows a 18-bit transmit mode example when PACKEN = 0. Note that when RGB_FMT_EN is set, the least significant bits of R, G, and B are dropped.

Table 15-29. Data Sent in 18-bit Transmit Mode with Packing Disabled

DMA Data	Pin Data (18-bits)	
	RGB_FMT_EN = 0	RGB_FMT_EN = 1
0x0123 4566	0x3 4567	0x0 8459
0x89AB CDEF	0x3 CDEF	0x2 ACFB
0x0123 4567	0x3 4567	0x0 8459

24-Bit Transmit Mode

For 24-bit transmit mode, if PACKEN = 1, the DMA is a 32-bit DMA and the EPPI will unpack three 32-bit words from memory into four 24-bit words to be transmitted out. The effect of the SWAPEN bit setting is shown in the table below.

Table 15-30 shows a 24-bit transmit mode example when PACKEN = 1:

Table 15-30. Data Sent in 24-bit Transmit Mode

DMA Data (32 bits)	Pin Data when SWAPEN = 0	Pin Data when SWAPEN = 1
$R_1B_0G_0R_0$	$B_0G_0R_0$	$R_0G_0B_0$
$G_2R_2B_1G_1$	$B_1G_1R_1$	$R_1G_1B_1$
$B_3G_3R_3B_2$	$B_2G_2R_2$	$R_2G_2B_2$
	$B_3G_3R_3$	$R_3G_3B_3$

8-Bit Split Transmit Mode

For 8-bit split transmit mode, PACKEN is not valid. The EPPI always unpacks the 32-bit DMA data into four bytes to be transmitted out.

Table 15-31 shows an 8-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 0 and SWAPEN = 0:

Table 15-31. Data sent in 8-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	**DMA1 DATA (32 bits)**	**Pin Data (8 bits)**	**DMA0 DATA (32 bits)**	**Pin Data (8 bits)**
$Y_3Y_2Y_1Y_0$	$U_1V_1U_0V_0$	V_0	$U_1V_1U_0V_0$	V_0
$Y_7Y_6Y_5Y_4$	$U_3V_3U_2V_2$	Y_0	$Y_3Y_2Y_1Y_0$	Y_0
		U_0	$U_3V_3U_2V_2$	U_0
		Y_1	$Y_7Y_6Y_5Y_4$	Y_1
		V_1		V_1
		Y_2		Y_2
		U_1		U_1
		Y_3		Y_3
		V_2		V_2
		Y_4		Y_4
		U_2		U_2
		Y_5		Y_5
		V_3		V_3
		Y_6		Y_6
		U_3		U_3
		Y_7		Y_7

Table 15-32 shows an 8-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 1 and SWAPEN = 0:

Table 15-32. Data Sent in 8-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (8 bits)	DMA0 DATA (32 bits)	Pin Data (8 bits)
$Y_3Y_2Y_1Y_0$	$V_3V_2V_1V_0$	V_0	$V_3V_2V_1V_0$	V_0
$Y_7Y_6Y_5Y_4$	$U_3U_2U_1U_0$	Y_0	$Y_3Y_2Y_1Y_0$	Y_0
	$V_7V_6V_5V_4$	U_0	$U_3U_2U_1U_0$	U_0
	$U_7U_6U_5U_4$	Y_1	$Y_7Y_6Y_5Y_4$	Y_1
		V_1		V_1
		Y_2		Y_2
		U_1		U_1
		Y_3		Y_3
		V_2		V_2
		Y_4		Y_4
		U_2		U_2
		Y_5		Y_5
		V_3		V_3
		Y_6		Y_6
		U_3		U_3
		Y_7		Y_7

Table 15-33 shows an 8-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 0 and SWAPEN = 1:

Table 15-33. Data Sent in 8-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (8 bits)	DMA0 DATA (32 bits)	Pin Data (8 bits)
$Y_3Y_2Y_1Y_0$	$U_1V_1U_0V_0$	U_1	$U_1V_1U_0V_0$	U_1
$Y_7Y_6Y_5Y_4$	$U_3V_3U_2V_2$	Y_3	$Y_3Y_2Y_1Y_0$	Y_3
		V_1	$U_3V_3U_2V_2$	V_1
		Y_2	$Y_7Y_6Y_5Y_4$	Y_2
		U_0		U_0
		Y_1		Y_1
		V_0		V_0
		Y_0		Y_0
		U_3		U_3
		Y_7		Y_7
		V_3		V_3
		Y_6		Y_6
		U_2		U_2
		Y_5		Y_5
		V_2		V_3
		Y_4		Y_4

Table 15-34 shows an 8-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 1 and SWAPEN = 1:

Table 15-34. Data Sent in 8-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (8 bits)	DMA0 DATA (32 bits)	Pin Data (8 bits)
$Y_3Y_2Y_1Y_0$	$V_3V_2V_1V_0$	V_3	$V_3V_2V_1V_0$	V_3
$Y_7Y_6Y_5Y_4$	$U_3U_2U_1U_0$	Y_3	$Y_3Y_2Y_1Y_0$	Y_3
	$V_7V_6V_5V_4$	U_3	$U_3V_3U_2V_2$	U_3
	$U_7U_6U_5U_4$	Y_2	$Y_7Y_6Y_5Y_4$	Y_2
		V_2		V_2
		Y_1		Y_1
		U_2		U_2
		Y_0		Y_0
		V_1		V_1
		Y_7		Y_7
		U_1		U_1
		Y_6		Y_6
		V_0		V_0
		Y_5		Y_5
		U_0		U_0
		Y_4		Y_4

10/12/14/16-Bit Split Transmit Mode with SPLT_16 = 0

For 16-bit split transmit mode, PACKEN is not valid. The EPPI always unpacks the 32-bit DMA data into two 16-bit words to be transmitted out. For 10-, 12-, or 14-bit split transmit modes, the EPPI first unpacks the data in the same way as for 16-bit transmit mode, but transmits only the required number of least significant bits.

Table 15-35 shows a 16-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 0 and SWAPEN = 0:

Table 15-35. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	**DMA1 DATA (32 bits)**	**Pin Data (16 bits)**	**DMA0 DATA (32 bits)**	**Pin Data (16 bits)**
Y_1Y_0	U_0V_0	V_0	U_0V_0	V_0
Y_3Y_2	U_1V_1	Y_0	Y_1Y_0	Y_0
		U_0	U_1V_1	U_0
		Y_1	Y_3Y_2	Y_1
		V_1		V_1
		Y_2		Y_2
		U_1		U_1
		Y_3		Y_3

Table 15-36 shows a 16-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 1 and SWAPEN = 0:

Table 15-36. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (16 bits)	DMA0 DATA (32 bits)	Pin Data (16 bits)
Y_1Y_0	V_1V_0	V_0	V_1V_0	V_0
Y_3Y_2	U_1U_0	Y_0	Y_1Y_0	Y_0
	V_3V_2	U_0	U_1U_0	U_0
	U_3U_2	Y_1	Y_3Y_2	Y_1
		V_1		V_1
		Y_2		Y_2
		U_1		U_1
		Y_3		Y_3

Table 15-37 shows a 16-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 0 and SWAPEN = 1:

Table 15-37. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (16 bits)	DMA0 DATA (32 bits)	Pin Data (16 bits)
Y_1Y_0	V_0U_0	V_0	V_0U_0	V_0
Y_3Y_2	V_1U_1	Y_1	Y_1Y_0	Y_1
		U_0	V_1U_1	U_0
		Y_0	Y_3Y_2	Y_0
		V_1		V_1
		Y_3		Y_3
		U_1		U_1
		Y_2		Y_2

Table 15-38 shows an 16-bit split transmit mode example with SPLT_EVEN_ODD = 1, SUBSPLT_ODD = 1 and SWAPEN = 1:

Table 15-38. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	DMA1 DATA (32 bits)	Pin Data (16 bits)	DMA0 DATA (32 bits)	Pin Data (16 bits)
Y_1Y_0	V_1V_0	V_1	V_1V_0	V_1
Y_3Y_2	U_1U_0	Y_1	Y_1Y_0	Y_1
	V_3V_2	U_1	U_1U_0	U_1
	U_3U_2	Y_0		Y_0
		V_0		V_0
		Y_3		Y_1
		U_0		U_0
		Y_2		Y_2

16-Bit Split Transmit Mode with SPLT_16 = 1

For 16-bit split transmit mode, PACKEN is not valid. The EPPI always unpacks the 32-bit DMA data into two 16-bit words to be transmitted out. The SPLT_16 bit is only valid when DLEN = 16 bits.

Table 15-39 shows a 16-bit split transmit mode example with SPLT_16 = 1, SUBSPLT_ODD = 0 and SWAPEN = 0:

Table 15-39. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
DMA0 DATA (32 bits)	**DMA1 DATA (32 bits)**	**Pin Data (16 bits)**	**DMA0 DATA (32 bits)**	**Pin Data (16 bits)**
$Y_3Y_2Y_1Y_0$	$U_1V_1U_0V_0$	V_0Y_0	$U_1V_1U_0V_0$	V_0Y_0
$Y_7Y_6Y_5Y_4$	$U_3V_3U_2V_2$	U_0Y_1	$Y_3Y_2Y_1Y_0$	U_0Y_1
		V_1Y_2	$U_3V_3U_2V_2$	V_1Y_2
		U_1Y_3	$Y_7Y_6Y_5Y_4$	U_1Y_3
		V_2Y_4		V_2Y_4
		U_2Y_5		U_2Y_5
		V_3Y_6		V_3Y_6
		U_3Y_7		U_3Y_7

Table 15-40 shows a 16-bit split transmit mode example with SPLT_16 = 1, SUBSPLT_ODD = 1 and SWAPEN = 0:

Table 15-40. Data Sent in 16-bit Split Transmit Mode

DMACFG = 1			DMACFG = 0	
PRIMARY DMA DATA (32 bits)	SECONDARY DMA DATA (32 bits)	Pin Data (16 bits)	DMA0 DATA (32 bits)	Pin Data (16 bits)
$Y_3Y_2Y_1Y_0$	$V_3V_2V_1V_0$	V_0Y_0	$V_3V_2V_1V_0$	V_0Y_0
$Y_7Y_6Y_5Y_4$	$U_3U_2U_1U_0$	U_0Y_1	$Y_3Y_2Y_1Y_0$	U_0Y_1
	$V_7V_6V_5V_4$	V_1Y_2	$U_3U_2U_1U_0$	V_1Y_2
	$U_7U_6U_5U_4$	U_1Y_3	$Y_7Y_6Y_5Y_4$	U_1Y_3
		V_2Y_4		V_2Y_4
		U_2Y_5		U_2Y_5
		V_3Y_6		V_3Y_6
		U_3Y_7		U_3Y_7

Programming Model

The following sections describe the EPPI programming model.

DMA Operation

The EPPI must be used with the processor's DMA engine. This section discusses how the two interact. For more information about the DMA engine, including default EPPI DMA channel mappings, see Chapter 7, “Direct Memory Access”.

The EPPI connects to the DMA channels in the following manner:

- EPPI0 always connects to DMA Channel 12 only
- EPPI1 and EPPI2 share DMA Channels 13 and 14. Each EPPI can connect to either or both of these DMA channels, depending on the mode of operation

This is shown visually in Figure 15-8 on page 15-28.

The EPPI DMA channels can be configured for either transmit or receive operation, and have a maximum throughput of (EPPIx_CLK) x (32 bits/transfer). In modes where data lengths permit, packing may be possible in order to increase transfer bandwidth. The highest throughput is achieved with 8-bit data and packing mode enabled.

Configuring the EPPI DMA channels is a necessary step toward using the EPPI 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 EPPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video is transferred, or 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 is transferred, and again when the whole frame is transferred.

Following is the general procedure for setting up DMA operation with the EPPI. For more information about configuring the DMA, see Chapter 7, "Direct Memory Access".

1. Configure the DMA registers as appropriate for the desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate EPPI registers.
4. Enable the EPPI by writing a 1 to bit 0 in EPPIx_CONTROL.

In addition, there are two sets of DMA Watermark levels to be programmed in the EPPIx_CONTROL register: Regular Watermark and Urgent Watermark. Two examples are given below: one showing the operation of the watermarks during transmit modes, the other showing their operation during receive modes.

For transmit modes

Let the urgent watermark be set to 25% Full (FIFO_UWM = b#11) and the regular watermark be set to 75% Full (FIFO_RWM = b#01). When the EPPI is enabled, the FIFO is initially empty. An urgent DMA request is asserted until the FIFO reaches urgent level, for example, the FIFO becomes 25% full. Then regular DMA requests are made until the FIFO becomes full. After that, the following things can happen (refer to Figure 15-9):

1. **State T0** Suppose, at the very beginning, before the EPPI has moved out the first data, that the FIFO is full. (Note that a full FIFO is not necessary to start moving data out, but is assumed here for simplicity) No DMA request.

2. **State T1** The EPPI has moved some data out and there are a few spaces in the FIFO. No DMA request.
3. **State T2** Because the data level is reduced to the regular watermark level, the DMA starts performing Regular DMA requests. This will result in the following two cases:
4. **State T3_0** Case 1. The DMA request is granted, and data is moved into the FIFO from L3. Regular DMA requests will stop when the FIFO is full. It returns to state T0.
5. **State T3_1** Case 2. The regular DMA Request is not granted. The EPPI continues moving data out and the data level continues to decrease.
6. **State T4** Because the data level is reduced to the urgent watermark level, the regular DMA request is changed to an Urgent DMA request. This will result in the following two cases:
7. **State T5_0** Case 1. The urgent DMA request is granted, and more data is moved into the FIFO from L3. When the data level has increased to the Regular Watermark level, the Urgent DMA request is changed to a regular DMA request. It returns to state T2.
8. **State T5_1** Case 2. The urgent DMA request is not granted, and the data level continues to decrease. When the EPPI has moved out all of the data, an underflow error occurs.

For transmit modes, the numerical value of the urgent watermark should be less than that of the regular watermark.

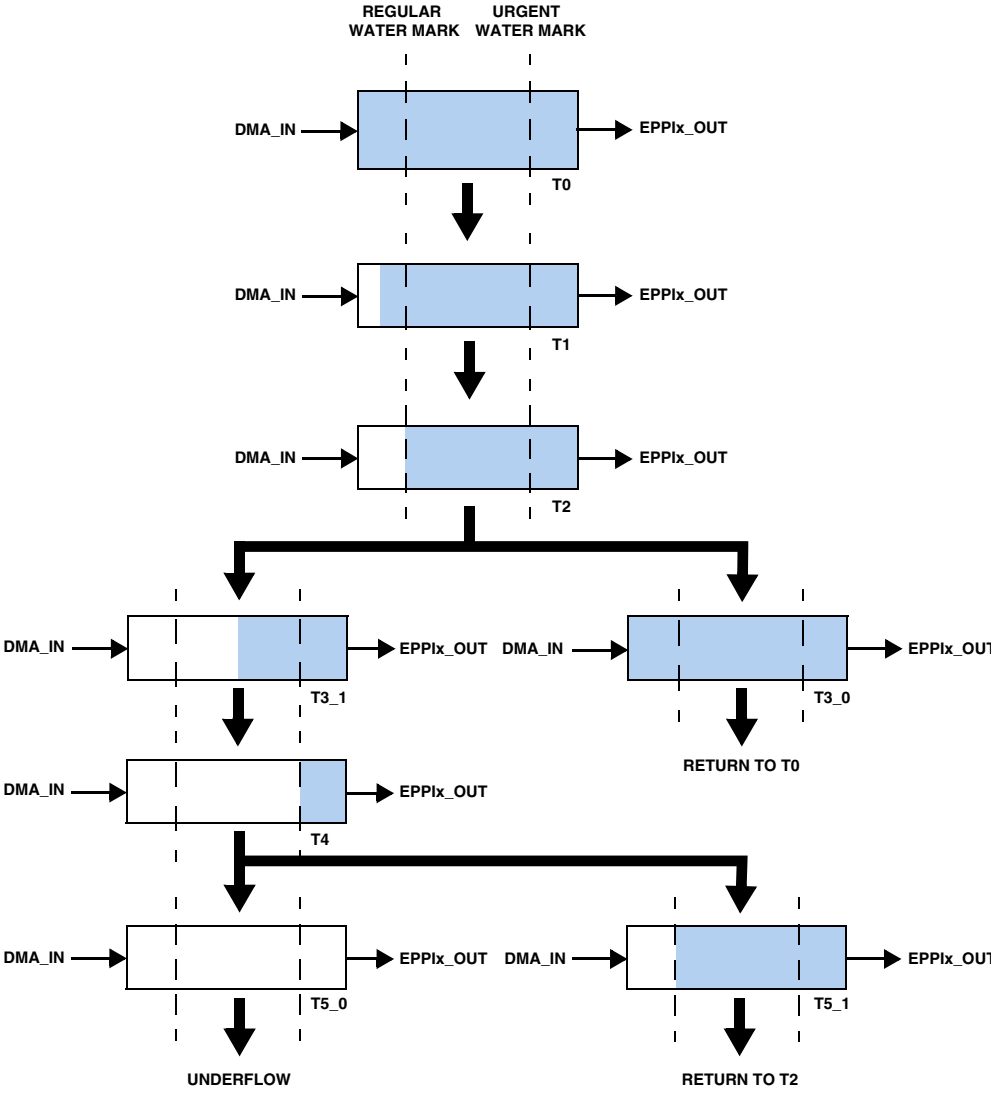


Figure 15-9. FIFO States during Transmit Modes

For receive modes

Let the urgent watermark be set to 75% Full (`FIFO_UWM` = b#01) and the regular watermark be set to 25% Full (`FIFO_RWM` = b#11). When the EPPI is enabled, the FIFO is initially empty. After that, the following things can happen (Refer to Figure 15-10):

1. **State T0** Suppose, at the very beginning, before the EPPI receives the first data, that the FIFO is empty. No DMA request.
2. **State T1** The EPPI has moved some data in and there are a few data in the FIFO. No DMA request.
3. **State T2** Because the data level has reached the regular watermark level, the DMA starts performing regular DMA requests. This will result in the following two cases:
4. **State T3_0** Case 1. The DMA request is granted, and data is moved out of the FIFO to L3. Regular DMA requests will stop when the FIFO is empty. It returns to state T0.
5. **State T3_1** Case 2. The regular DMA request is not granted. The EPPI continues moving data in and the data level continues to increase.
6. **State T4** Because the data level has reached the urgent watermark level, the regular DMA request is changed to an Urgent DMA request. This will result in one of the following two cases:
7. **State T5_0** Case 1. The urgent DMA request is granted, and more data is moved out of the FIFO to L3. When the data level has decreased to the regular watermark level, the urgent DMA request is changed to a regular DMA request. It returns to state T2.
8. **State T5_1** Case 2. The urgent DMA request is not granted, and the data level continues to increase. After the EPPI has filled all of the available space in the FIFO, an overflow error occurs.

For receive modes, the value of the Regular Watermark should be less than that of the Urgent Watermark.

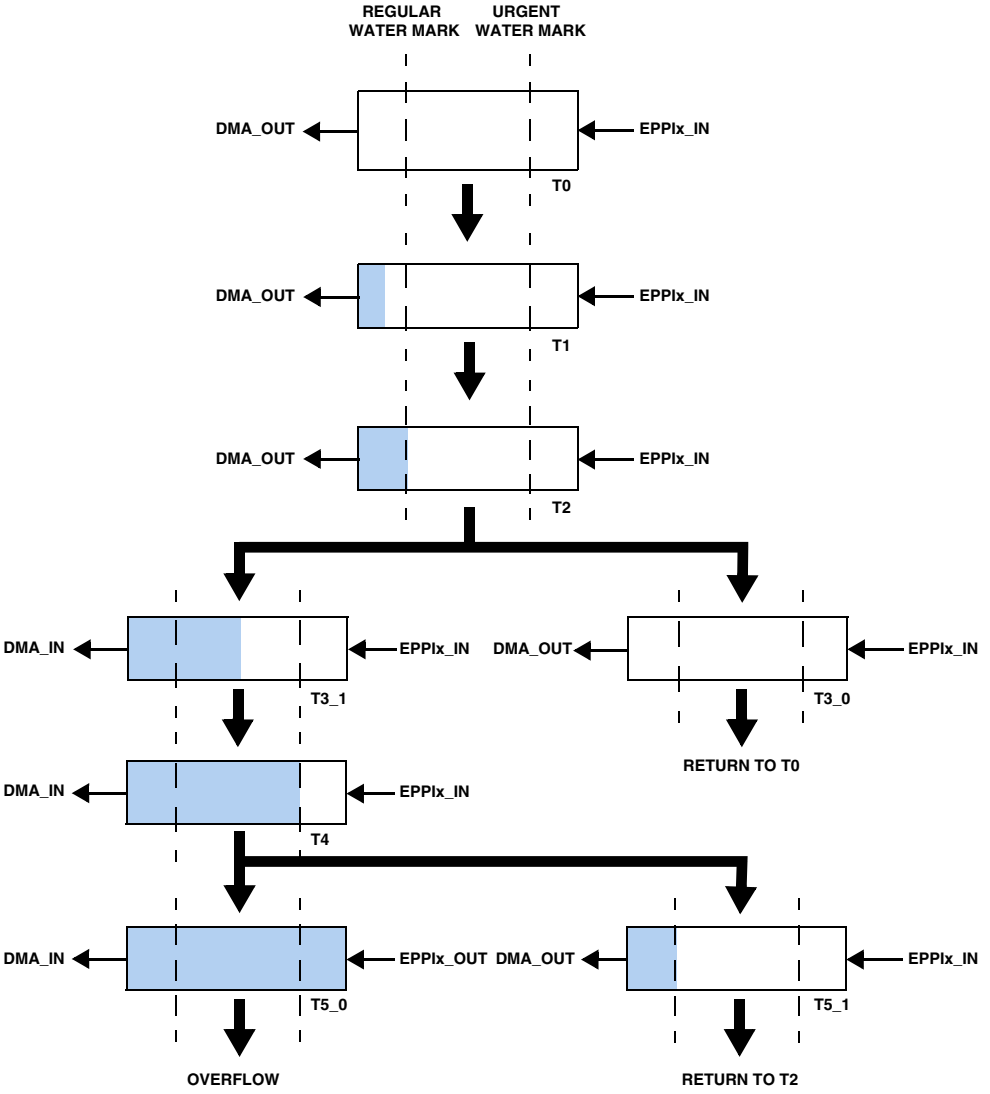


Figure 15-10. FIFO States during Receive Modes

Note the following:

- For transmit modes with 1, 2 or 3 frame syncs, the EPPI will not start transmitting data until its FIFO has some valid data to transmit. Therefore, the frame syncs should be sent only some time after enabling the EPPI, so that by then, the EPPI FIFO contains sufficient data.
- For transmit modes with internal frame syncs, the EPPI will not start generating frame syncs until its FIFO is full.
- For GP 0 FS TX modes and for ITU-R 656 Output mode (BLANKGEN = 1), the EPPI will not start transmitting data until the EPPI FIFO becomes full.
- When using two DMA channels (DMACFG = 1), both FIFO should be full.

Elevating EPPI Urgent Requests at DDR Controller Interface

In addition to the Urgent watermarks that control the priority of EPPI, two control bits are available to elevate the priority of EPPI0, EPPI1, and EPPI2 transactions at the DDR controller interface.

Lower priority resources can typically gain access to the DDR interface, due to the pipelined nature of the requests at the DDR interface, and due to the DEB bus submitting a DDR request at a maximum of every other SCLK cycle.

The CORE_EPPI_PRIO and SYS_EPPI_PRIO bits in the HMDMA0_CONTROL register (see "Handshake MDMA Control Registers" on page 7-113) are provided to ensure that under EPPI urgent conditions, for more efficient use of the external memory bus bandwidth, only the EPPI can gain access

to the DDR memory. These bits are required only under high DDR activity when the core and several other DMA channels (including EPPI-DMA channels) simultaneously access the DDR memory.

Setting the CORE_EPPI_PRIO bit in the HMDMA0_CONTROL register, blocks all core accesses to the DDR memory as long as any EPPI request stays urgent, or for a maximum period of 124 system clock cycles. After 124 system clock cycles, the core can gain access for a period of 4 system clock cycles.

Setting the SYS_EPPI_PRIO bit in the HMDMA0_CONTROL register blocks all DMA channels in DMAC0, as well as the USB, PIXC and DMAC1 MDMA channels—as long as any EPPI request stays urgent.

Note that while the EPPI request stays urgent, other peripherals on DMAC1 are not blocked and any unused bandwidth is allocated to the DMA channels on DMAC1 based on their priority levels.

Also note that this feature can be individually enabled for EPPI0, EPPI1, or EPPI2 by enabling the urgent watermark in the respective EPPI control register.

As an exception, under all conditions, a TESTSET instruction has higher priority than EPPI-DMA urgent requests. Therefore, even when the CORE_EPPI_PRIO bit of the HMDMA0_CONTROL register is set, a TESTSET instruction will not be blocked under an EPPI-DMA urgent condition.

Also, there may be an increase in the interrupt service latency when core accesses to the DDR are blocked due to pending urgent EPPI accesses.

If the DMA is used in descriptor mode, the DMA descriptors should be placed in L2 or L1 memory if EPPI urgent conditions are being seen.

System Configuration

Due to pin muxing, there are restrictions on the possible system configurations of the EPPI channels. Table 15-41 shows the possible system configurations.

Table 15-41. EPPI System Configurations

EPPI 0	EPPI 1	EPPI 2
8-24 bits	*Not supported*	*Not supported*
8-18 bits	8 bits	8 bits
8-18 bits	10-14 bits	*Not supported*
8-18 bits	8 bits	*Not supported*
8-18 bits	16 bits	*Not supported*
8-24 bits	*Not supported*	8 bits

In addition, Split mode may be used with EPPI1 or EPPI2 in the last three configurations. This is done by setting the EPPI's `DMACFG` bit, but is only valid if the `SPLT_EVEN_ODD` bit is also set.

EPPI Registers

Table 15-42 contains a list of EPPI memory-mapped registers (MMRs). Default values of all MMRs are 0x0, except `EPPIx_CLIP` whose default value is 0xFF00 FF00.

Table 15-42. EPPI Memory-Mapped Registers

Address	Register Name	Width	Description
0xFFC0 1000	EPPIx_STATUS	16	"EPPI Status (EPPIx_STATUS) Register" on page 15-77
0xFFC0 1004	EPPIx_HCOUNT	16	"EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT)" on page 15-93
0xFFC0 1008	EPPIx_HDELAY	16	"EPPI Horizontal Delay Register (EPPIx_HDELAY)" on page 15-92
0xFFC0 100C	EPPIx_VCOUNT	16	"EPPI Vertical Transfer Count Register (EPPIx_VCOUNT)" on page 15-91
0xFFC0 1010	EPPIx_VDELAY	16	"EPPI Vertical Delay Register (EPPIx_VDELAY)" on page 15-91
0xFFC0 1014	EPPIx_FRAME	16	"EPPI Lines per Frame Register (EPPIx_FRAME)" on page 15-90
0xFFC0 1018	EPPIx_LINE	16	"EPPI Samples per Line Register (EPPIx_LINE)" on page 15-90
0xFFC0 101C	EPPIx_CLKDIV	16	"EPPI Clock Divide Register (EPPIx_CLKDIV)" on page 15-93
0xFFC0 1020	EPPIx_CONTROL	32	"EPPIx Control (EPPIx_CONTROL) Register" on page 15-80
0xFFC0 1024	EPPIx_FS1W_HBL	32	"EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL)" on page 15-94
0xFFC0 1028	EPPIx_FS1P_AVPL	32	"EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL)" on page 15-96
0xFFC0 102C	EPPIx_FS2W_LVB	32	"EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB)" on page 15-95
0xFFC0 1030	EPPIx_FS2P_LAVF	32	"EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF)" on page 15-97
0xFFC0 1034	EPPIx_CLIP	32	"EPPI Clipping Register (EPPIx_CLIP)" on page 15-98

The MMR addresses shown in Table 15-42 on page 15-74 refer to the EPPIx registers, that start at a base address of 0xFFC0 1000. EPPI1 and EPPI2 have base addresses of 0xFFC0 1300 and 0xFFC0 2900, respectively, and follow the same register address increments as shown above for EPPIx.

Table 15-43 shows which of the MMRs are valid for which operating modes (an "X" indicates that the register is valid for the particular mode):

Table 15-43. MMR Usage Modes

MMR	GP 1 Frame Sync Modes				GP 2 Frame Sync Modes				GP 3 Frame Sync Modes				GP 0 Frame Sync Modes				ITU RX Modes
	Ext FS		Int FS		Ext FS		Int Fs		Ext FS		Int FS		Ext Trig		Int Trig		
	RX	TX	RX	TX	RX	TX	RX	TX	RX	TX	RX	TX	RX	TX	RX	TX	
EPPIx_FRAME					X	X	X	X	X	X	X	X					X
EPPIx_LINE	X	X	X	X	X	X	X	X	X	X	X	X					X
EPPIx_HDELAY	X	X	X	X	X	X	X	X	X	X	X	X					
EPPIx_HCOUNT	X	X	X	X	X	X	X	X	X	X	X	X					
EPPIx_VDELAY					X	X	X	X	X	X	X	X					
EPPIx_VCOUNT						X	X	X	X	X	X	X					
EPPIx_FS1W_HBL			X	X			X	X			X	X					
EPPIx_FS1P_AVPL			X	X			X	X			X	X					
EPPIx_FS2W_LVB							X	X			X	X					
EPPIx_FS2P_LAVF							X	X			X	X					

EPPIx_CLKDIV is valid for all modes when an internal clock is used (ICLKEN = 1 in EPPIx_CONTROL).

EPPIx_CLIP is valid for all transmit modes with an 8-bit or 16-bit data lengths.

The following registers have multiplexed operation. In GP 1/2/3 Frame Sync modes, they are used for generation of EPPIx_FS1/EPPIx_FS2. In GP 0 FS transmit mode when BLANKGEN = 1, they are used as internal blanking generation registers:

EPPIx_FS2_WIDTH = EPPIx_LVB (Lines of Vertical Blanking)

EPPIx_FS2_PERIOD = EPPIx_LAVF (Lines of Active Video per Field)

EPPIx_FS1_WIDTH = EPPIx_HBL (Horizontal Blank Samples per Line)

EPPIx_FS1_PERIOD = EPPIx_AVPL (Active Video Samples per Line)

Each pair of the above registers has the same physical address.

EPPI Status (EPPIx_STATUS) Register

The EPPIx_STATUS register, shown in Figure 15-11, is a 16-bit register that indicates the status of the EPPI.

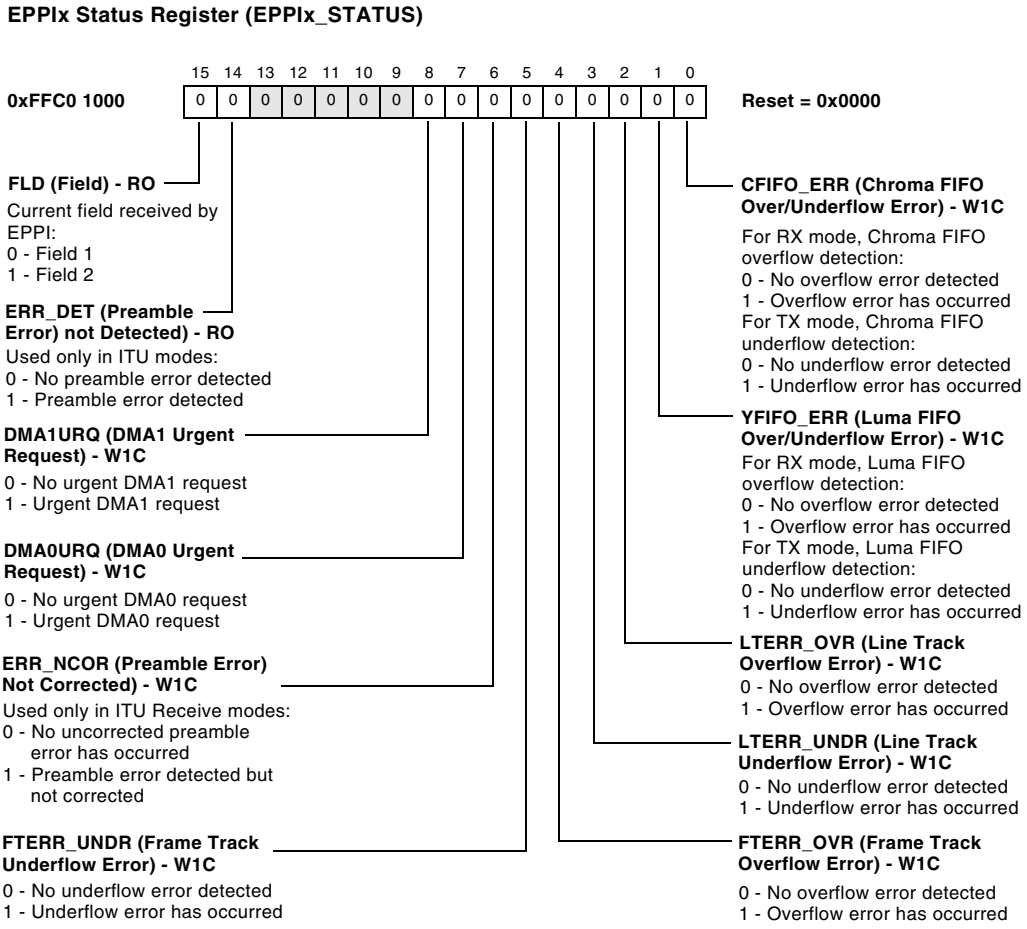


Figure 15-11. EPPI Status Register

CFIFO_ERR (Chroma FIFO Overflow/Underflow Error)

When set, this bit indicates that the Chroma FIFO has overflowed (in receive mode) or underflowed (in transmit mode). This bit is sticky and must be cleared in software by writing 1 to it.

In transmit mode, the CFIFO_ERR is set to indicate an underflow condition only when DMACFG in EPPIx_CONTROL is set.

YFIFO_ERR (Luma FIFO Overflow/Underflow Error)

When set, this bit indicates that the Luma FIFO has overflowed (in receive mode) or underflowed (in transmit mode). This bit is sticky and must be cleared in software by writing 1 to it.

When in transmit mode, a 1 in YFIFO_ERR or CFIFO_ERR indicates that the FIFOs have underflowed. However, the EPPI may still be transmitting data out the pins. Therefore, to avoid incomplete data transmission, the EPPI should not be disabled immediately after observing a 1 value in these bit. The time delay necessary depends on the EPPI clock and on the EPPI data length.

LTERR_OVR (Line Track Overflow)

This bit indicates whether a Line Track Overflow Error has occurred (if set, = 1) or not (if clear, = 0). This bit is sticky and must be cleared in software by writing 1 to it.

LTERR_UNDR (Line Track Underflow)

This bit indicates whether a Line Track Underflow Error has occurred (if set, = 1) or not (if clear, = 0). This bit is sticky and must be cleared in software by writing 1 to it.

FTERR_OVR (Frame Track Overflow)

This bit indicates whether a Frame Track Overflow Error has occurred (if set, = 1) or not (if clear, = 0). This bit is sticky and must be cleared in software by writing 1 to it.

FTERR_UNDR (Frame Track Underflow)

This bit indicates whether a Frame Track Underflow Error has occurred (if set, = 1) or not (if clear, = 0). This bit is sticky and must be cleared in software by writing 1 to it

ERR_NCOR (Preamble Error not Corrected)

This bit is useful only in the ITU receive modes and indicates if an error in the status word of EAV or SAV sequences can not be cleared (if set, = 1) or not (if clear, = 0). This bit is sticky and must be cleared in software by writing 1 to it.

DMA1URQ (DMA1 Urgent Request)

This bit if set indicates that the EPPI is making an Urgent DMA Request. If the PAB writes a 1 to this bit, it is cleared and the DMA Urgent Request will go low in the next cycle.

DMA0URQ (DMA0 Urgent Request)

This bit if set indicates that the EPPI is making an Urgent DMA Request. If the PAB writes a 1 to this bit, it is cleared and the DMA Urgent Request will go low in the next cycle.

ERR_DET (Preamble Error Detected)

This bit is useful only in ITU receive modes and indicates if an error is detected in the status word of EAV or SAV sequences (if set, = 1) or not (if clear, = 0).

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 an EPPI error interrupt, unless this condition is masked off in the SICx_IMASK register.

FLD (Field)

This bit indicates if the current field being transferred is Field 1 (if clear, = 0) or Field 2 (if set, = 1)

EPPIx Control (EPPIx_CONTROL) Register

The EPPIx_CONTROL register, shown in Figure 15-12 and Figure 15-13, is a 32-bit register that configures the EPPI for operating mode, control signal polarities, and data width of the port.

Be aware that Figure 15-12 and Figure 15-13 split the EPPIx_CONTROL register "halves" across nonstandard boundaries in order to maintain the DLEN field intact.

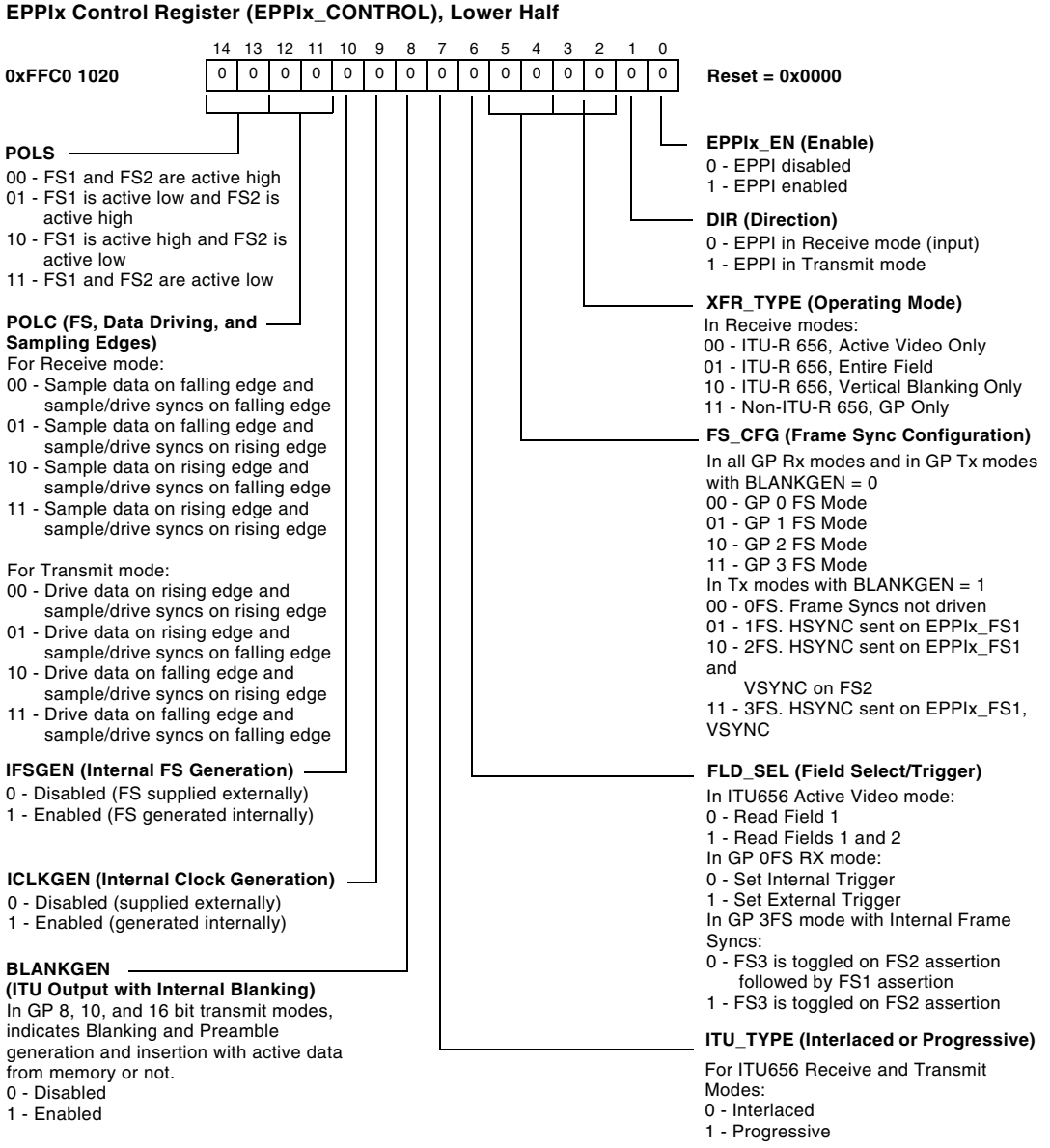


Figure 15-12. EPPIx Control Register, Lower Half

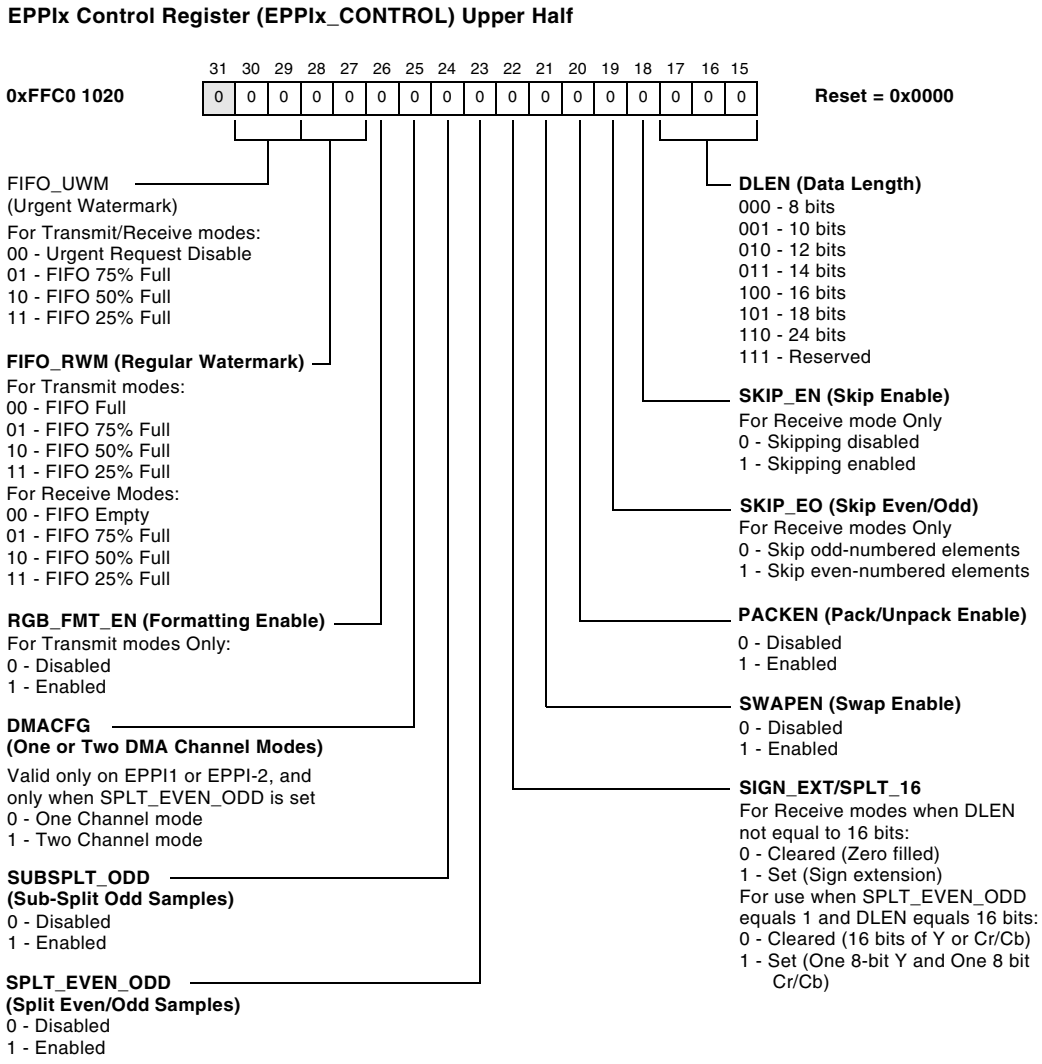


Figure 15-13. EPPIx Control Register, Upper Half

The EPPIx_EN when set, enables the EPPI for operation. On disabling the EPPI by writing a 0 to the EN bit of EPPIx_CONTROL, all EPPI MMRs, except EPIx_STATUS, do not return to their reset values. EPPI Interrupt and DMA requests go inactive. Internally generated EPPIx_CLK and Frame Syncs are aborted on disabling the EPPI.

Once the EPPI is enabled, none of the MMRs should be changed. If any change is required, the EPPI should first be disabled and then re-enabled after re-programming the MMRs.

DIR (Direction): Setting this bit configures the EPPI to transmit data. In transmit mode, data is moved out from memory through the EPPI. Clearing DIR configures the EPPI to receive data. In receive mode, data is captured by EPPI and moved to memory.

XFR_TYPE[1:0] (Operating Mode): The XFR_TYPE[1:0] field configures the EPPI for various modes of operation in receive mode. Programming XFR_TYPE with b#00, b#01, or b#10 configures the EPPI to receive data in ITU-R 656 active video only, entire field, or vertical blanking only modes respectively. Programming XFR_TYPE with 0x11 configures EPPI to operate in general purpose mode.

FS_CFG[1:0] (Frame Sync Configuration): The FS_CFG field is used to configure the frame syncs of the EPPI.

In receive modes and in transmit modes with BLANKGEN set to 1, setting this field to b#00, b#01, b#10, or b#11 configures EPPI for general purpose 0, 1, 2, or 3 frame sync modes respectively.

In transmit modes with BLANKGEN cleared to 0, a value of b#00 in this field means that frame syncs are not driven. A value of 0x01 means that HSYNC is driven on EPPIx_FS1. A value of b#10 means that HSYNC is driven on EPPIx_FS1 and that VSYNC is driven on EPPIx_FS2. A value of b#11 means that HSYNC is driven on EPPIx_FS1, that VSYNC is driven on EPPIx_FS2, and that FIELD is driven on EPPIx_FS3.

FLD_SEL (Field Select/Trigger): This bit is useful only in the ITU656 Active Video Only Mode and GP 0 FS RX Mode and GP 3 FS Mode with Internal Frame Syncs.

In ITU656 Active Video Only Mode, this indicates whether only Field 1 is received (if cleared, = 0) or both Field1 and Field2 are received (if set, = 1).

In GP 0 FS RX Mode, this indicates whether the trigger is external (if set, = 1) or internal (if cleared, = 0).

In GP 3 FS Mode with Internal Frame Syncs, this bit indicates, if the EPPIx_FS3 is toggled on every assertion of EPPIx_FS2 (if set, = 1) or if the EPPIx_FS3 is toggled on every EPPIx_FS1 assertion followed by EPPIx_FS2 assertion (if cleared, = 0)

ITU_TYPE (ITU Interface or Progressive): This bit is useful only for ITU receive modes. It indicates whether the ITU656 video is Interlaced (if cleared, = 0) or Progressive (if set, = 1)

BLANKGEN (ITU Output with Internal Blanking): This bit is useful in GP Transmit Mode when the data length is configured for 8-, 10-, or 16-bits. BLACKGEN specifies whether or not to generate blanking and preamble data and to insert it with the active data being transmitted from memory. If set, blanking and preamble data is generated and inserted with the active data. If cleared, the active data is transmitted from memory as is.

Frame syncs may be driven out along with the data based on the configurations of BLANKGEN and FS_CFG.

ICLKGEN (Internal Clock Generation): This bit indicates if the EPPIx_CLK is generated internally (if set, = 1) or is supplied by an external device (if cleared, = 0)

IFSGEN (Internal Frame Sync Generation): This bit indicates if the Frame Syncs are generated internally (if set, = 1) or are supplied by an external device (if cleared, = 0)

POLC[1:0] & POLS[1:0] (Clock Polarity and Frame Sync Polarity): The POLC[1:0] and POLS[1:0] bits allow the selection of the active level of the frame syncs and the sampling/driving edge of the EPPI clock, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities.

DLEN[2:0] (Data Length): The DLEN[2:0] field is programmed to specify the data width of the EPPI module. Note that due to pin muxing, there are restrictions on the possible system configurations of the EPPI channels. Table 15-41 on page 15-73 shows the possible configurations. In ITU-R 656 modes, the DLEN field should be configured for 8- or 10-bit width.

SKIP_EN (skip enable, bit 18)

For receive modes, if this bit is set, alternate even or odd data elements being read through the EPPI may be skipped based on the value programmed in the SKIP_EO bit.

SKIP_EO (skip even/odd, bit 19)

This bit is meaningful only in receive mode and when SKIP_EN is set. When SKIP_EO is zero, the odd numbered elements are skipped. When SKIP_EO is one, the even numbered elements are skipped. Element numbering starts from 1. Hence, when SKIP_EO is not set, the first incoming element is skipped, the third incoming element is skipped, and so on. 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 EPPI 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 while the other (whose SKIP_EO bit is set differently from the first processor's) could handle Chroma processing.

PACKEN/UNPACKEN (packing/unpacking enable, bit 20)

For receive modes this bit indicates if packing is enabled or not. For transmit modes this indicates if unpacking is enabled or not. DMA is always 32 bits wide if this bit is set. If this bit is not set and the DLEN is less than or equal to 16 bits, then the DMA is 16 bits wide.

For receive modes, if this bit is set, then the EPPI packs the incoming data into 32-bit words. If this bit is cleared, then the EPPI does not do any packing.

For transmit modes, if this bit is set, then the EPPI always unpacks the 32-bit data from DMA. If this bit is not set, the EPPI does not do any unpacking.

SWAPEN (swap enable, bit 21)

For receive modes, the EPPI puts the first data in the most significant bits (if set, = 1) or puts the first data in the least significant bits (if cleared, = 0) of the DMA word.

For transmit modes, the EPPI transmits the most significant bits in the DMA word as the first data (if set, = 1) or transmits the least significant bits in the DMA word as the first data (if cleared, = 0).

SIGN_EXT/SPLT_16 (sign extension or zero filled, bit 22)

This bit has two different functions. When DLEN is not equal to 16 bits it acts as SIGN_EXT, and when DLEN is equal to 16 bits it acts as SPLT_16.

As SIGN_EXT, this bit is useful only for receive modes and indicates if the data is sign extended (if set, = 1) or zero filled (if cleared, = 0). This is valid only for data lengths of 8, 10, 12, 14, 18 or 24 bits.

As SPLT_16, this bit is useful only when SPLT_EVEN_ODD = 1 and DLEN = 16. If set (= 1), then this bit indicates that the 16-bit 4:2:2 YCrCb data has one 8-bit Y and one 8-bit of either Cr or Cb packed together. Note that Y is on bits 7:0 and Cr/Cb on bits 15:8. If cleared (= 0), then this bit indicates that the 16 bits of 4:2:2 YCrCb data has 16 bits of Y or Cr/Cb.

SPLT_EVEN_ODD (Split Even and Odd Data Samples)

If it is set, EPPI will split even and odd samples. See "Split Receive Modes" on page 15-31 and "Split Transmit Modes" on page 15-31 for more details.

SUBSPLT_ODD (Sub-Split Odd Samples)

If it is set, EPPI will sub-split odd samples. It is valid only if SPLT_EVEN_ODD is set.

DMACFG (One or Two DMA Channels Mode)

If it is set, EPPI will use two DMA Channels, else EPPI will use only one DMA Channel. It is valid only when SPLT_EVEN_ODD is set.

RGB_FMT_EN (RGB Formatting Enable)

This bit is valid only for 16-bit or 18-bit transmit modes. For 18-bit transmit modes, if this is set EPPI converts the RGB888 from Memory into RGB666 output data. For 16-bit transmit modes, if this is set, EPPI converts RGB888 from Memory into RGB565 output data.

FIFO_RWM (FIFO Regular Watermarks) and FIFO_UWM (FIFO Urgent Watermarks)

These bits indicate the regular and the urgent watermark level for the FIFO respectively.

RGB_FMT_EN (RGB formatting enable, bit 26)

This bit is valid only for 16-bit or 18-bit transmit modes. For 18-bit transmit modes, if this bit is set, the EPPI converts the RGB888 data from memory into RGB666 output data. For 16-bit transmit modes, if this bit is set, the EPPI converts RGB888 data from memory into RGB565 output data.

`SPLT_EVEN_ODD` and `RGB_FMT_EN` should never be set simultaneously.

Windowing Registers

Windowing is a useful feature for applications where the region of interest is smaller than the active video stream (for example, sensor calibration, auto-focusing, etc.). It can result in significant DMA bandwidth reduction. Each EPPI supports windowing for GP Input modes and has six MMRs that are used to define the video frame. A pictorial view of these registers is shown in Figure 15-14.

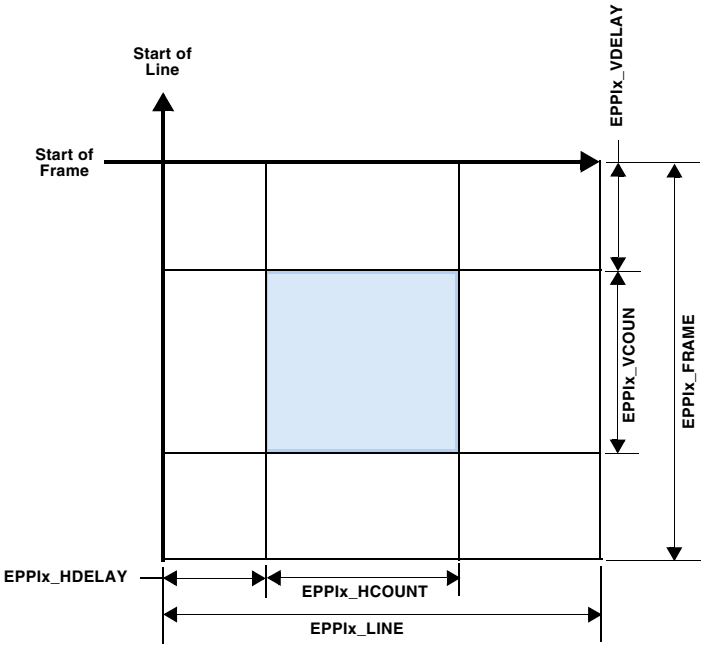


Figure 15-14. Windowing Registers to define a Frame

The shaded portion in Figure 15-14 is the captured/transmitted data. Note that windowing is valid for GP receive and transmit modes.

It is the user's responsibility to ensure the following:

```
EPPIx_VDELAY + EPPIx_VCOUNT <= EPPIx_FRAME
EPPIx_HDELAY + EPPIx_HCOUNT <= EPPIx_LINE
```

EPPI Lines per Frame Register (EPPIx_FRAME)

The EPPIx_FRAME register, shown in Figure 15-15, is a 16-bit register used to keep track of Frame Track Overflow and Underflow errors. It should be programmed with the number of lines expected per frame. Any write to the EPPIx_FRAME register will also write the same value to the EPPIx_VCOUNT register. However, any write to EPPIx_VCOUNT does not affect the EPPIx_FRAME register value. Therefore, the EPPIx_FRAME register should be programmed before the EPPIx_VCOUNT register.

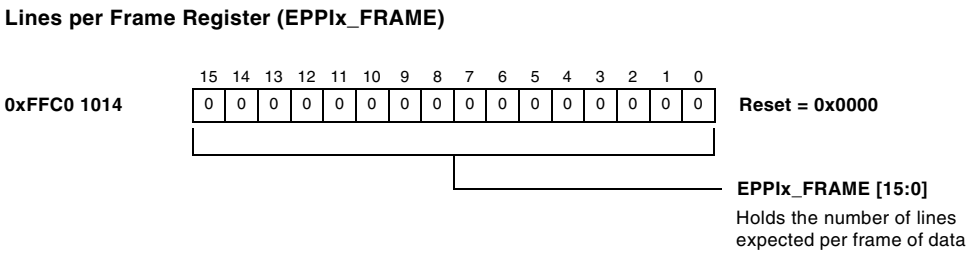


Figure 15-15. EPPI Lines per Frame Register

EPPI Samples per Line Register (EPPIx_LINE)

The EPPIx_LINE register, shown in Figure 15-16, is a 16-bit register used to keep track of Line Track Overflow and Underflow Errors. It should be programmed with the number of samples expected per line. Any write to the EPPIx_LINE register will also write the same value to the EPPIx_HCOUNT register. However, any write to EPPIx_HCOUNT does not affect the EPPIx_LINE register value. Therefore, the EPPIx_LINE register should be programmed before the EPPIx_HCOUNT register.

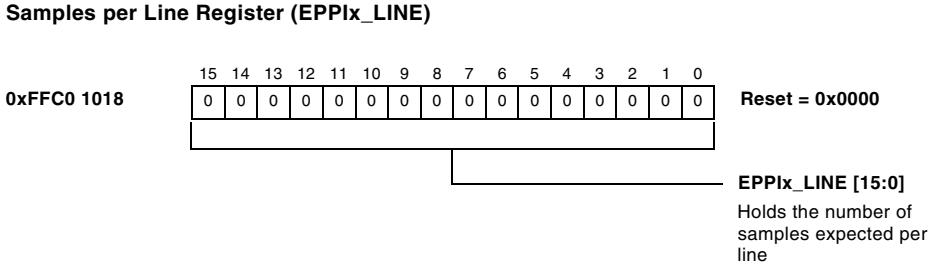


Figure 15-16. EPPI Samples per Line Register

EPPI Vertical Delay Register (EPPIx_VDELAY)

The EPPIx_VDELAY register, shown in Figure 15-17, is a 16-bit register and contains the number of lines to wait after the start of a new frame before starting to read/transmit data.

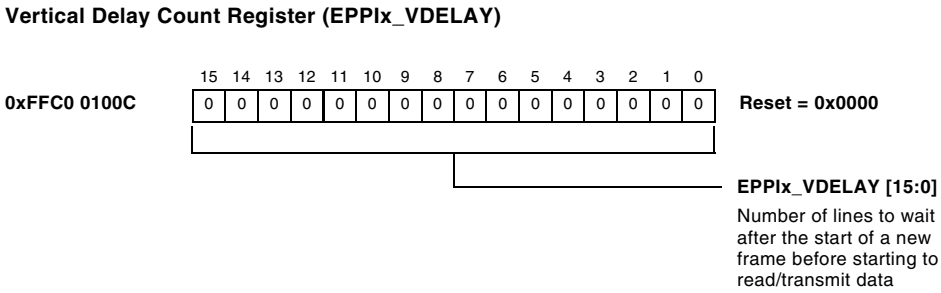


Figure 15-17. EPPI Vertical Delay Count Register

EPPI Vertical Transfer Count Register (EPPIx_VCOUNT)

The EPPIx_VCOUNT register, shown in Figure 15-18, is a 16-bit register and holds the number of lines to read in or write out, after EPPIx_VDELAY number of lines from the start of frame. Any write to the EPPIx_FRAME register modifies the EPPIx_VCOUNT register. However, any write to EPPIx_VCOUNT

does not affect the EPPIx_FRAME register value. Therefore, the EPPIO_VCOUNT register should be programmed after the EPPIx_FRAME register.

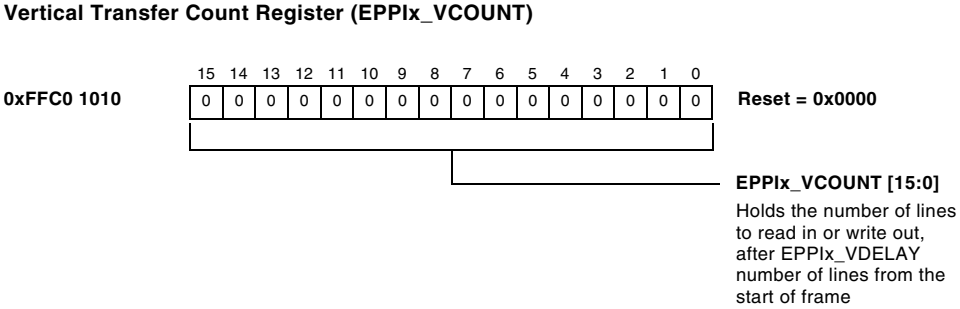


Figure 15-18. EPPI Vertical Transfer Count Register

EPPI Horizontal Delay Register (EPPIx_HDELAY)

The EPPIx_HDELAY register, shown in Figure 15-19, is a 16-bit register and contains the number of clock cycles to delay after the assertion of EPPIx_FS1 is detected before starting to read or write data.

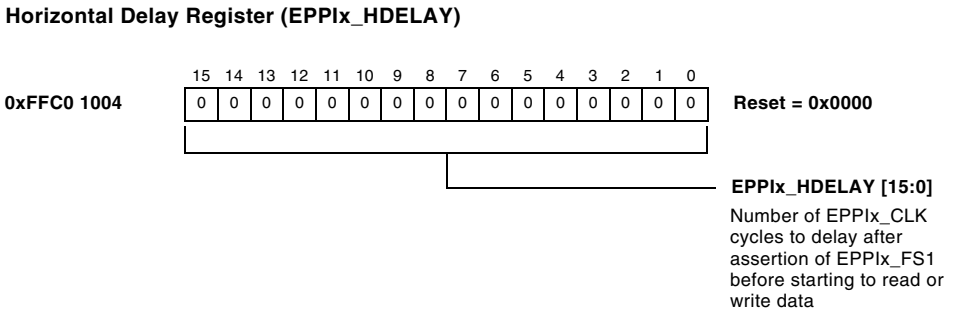


Figure 15-19. EPPI Horizontal Delay Register

EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT)

The EPPIx_HCOUNT register, shown in Figure 15-20, is a 16-bit register and holds the number of samples to read in or write out per line, after EPPIx_HDELAY number of cycles have expired since the assertion of EPPIx_FS1. Any write to the EPPIx_LINE register modifies the EPPIx_HCOUNT register. However, any write to EPPIx_HCOUNT does not affect the EPPIx_LINE register value. Therefore, the EPPIx_HCOUNT register should be programmed after the EPPIx_LINE register.

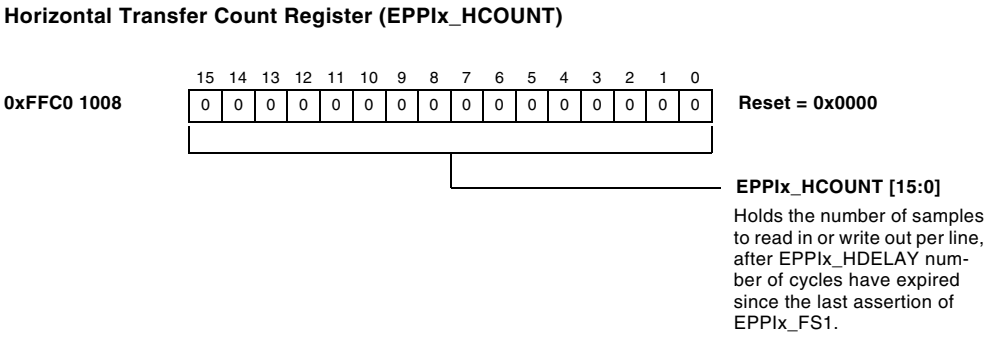


Figure 15-20. EPPI Horizontal Transfer Count Register

EPPI Clock Divide Register (EPPIx_CLKDIV)

The EPPIx_CLKDIV register, shown in Figure 15-21, is a 16-bit register used for internal clock generation. The generated clock frequency is given by following formula:

```
EPPIx_CLK = (SCLK) / (2 * (EPPIx_CLKDIV[15:0] + 1))
```

Note that a value of 0xFFFF is invalid for EPPIx_CLKDIV register.

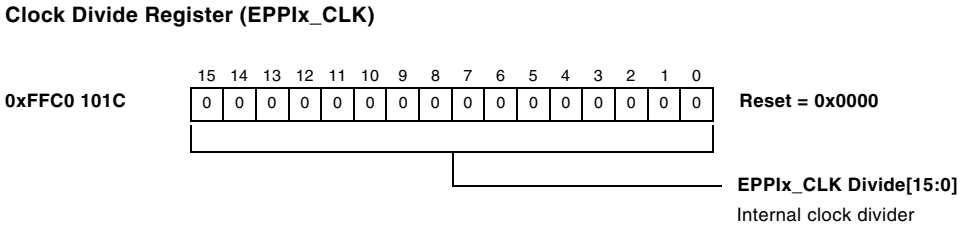


Figure 15-21. EPPI Clock Divide Register

Frame Sync/ Blanking Generation Registers

The following sections describe the sync and blanking generation registers.

EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL)

The EPPIx_FS1W_HBL register, shown in Figure 15-22, is a 32-bit register.

In GP 1, 2 or 3 FS modes, it is used for the generation of Frame Sync 1. It contains the width required for FS1. The reference clock is EPPIx_CLK.

In GP Transmit mode with BLANKGEN = 1 in EPPIx_CONTROL, it contains the number of samples of horizontal blanking per line.

When used for blanking generation, only the lower 16 bits are valid.

A value of 0 for this register is illegal. If it is programmed as 0, the EPPI will regard its value as 1.

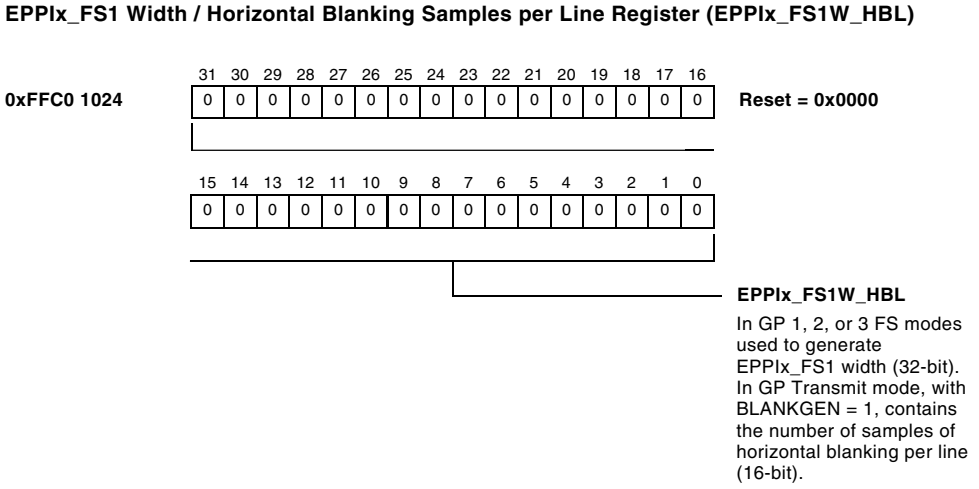


Figure 15-22. EPPI FS1 Width/Horizontal Blanking Samples per Line Register

EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB)

`EPPIx_FS2W_LVB`, shown in Figure 15-23, is a 32-bit register.

In GP 2 or 3 FS modes, it is used for the generation of Frame Sync 2. It contains the width required for FS2. The reference clock is `EPPIx_CLK`.

In GP Transmit mode with `BLANKGEN` = 1 in `EPPIx_CONTROL`, it contains the number or lines of vertical blanking.

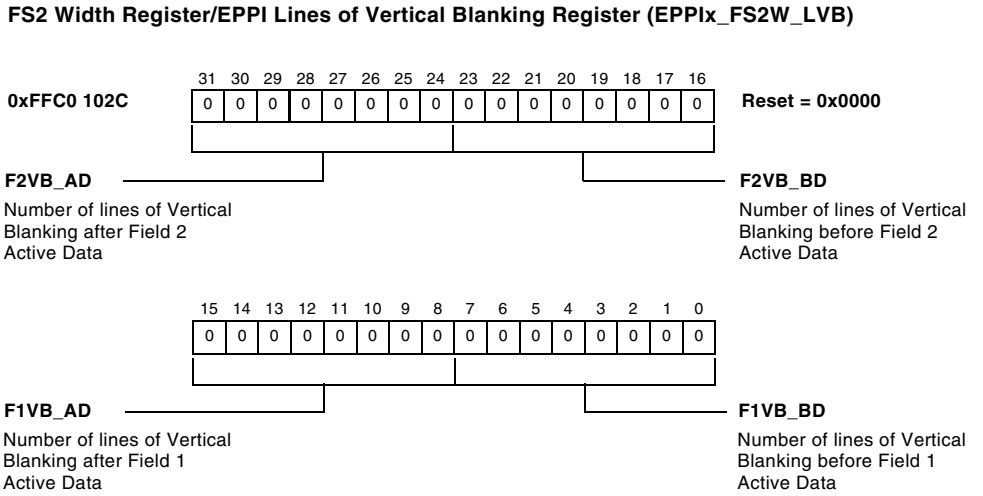


Figure 15-23. EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register

For progressive video, `F2VB_BD` and `F2VB_AD` are ignored.

A value of 0 in any of the fields is illegal. If programmed as 0, the EPPI regards its value as 1.

EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL)

The `EPPIx_FS1P_AVPL` register, shown in Figure 15-24, is a 32-bit register.

In GP 1, 2, or 3 FS modes, it is used for the generation of Frame Sync 1. It contains the period required for `EPPIx_FS1`. The reference clock is `EPPIx_CLK`.

In GP Transmit mode with `BLANKGEN` = 1 in `EPPIx_CONTROL`, it contains the number of samples of active video or vertical blanking samples per line. When used for blanking generation, only the lower 16 bits are valid.

A value of 0 for this register is illegal. If it is programmed as 0, the EPPI will regard its value as 1.

EPPIx_FS1 Period Register / EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPI

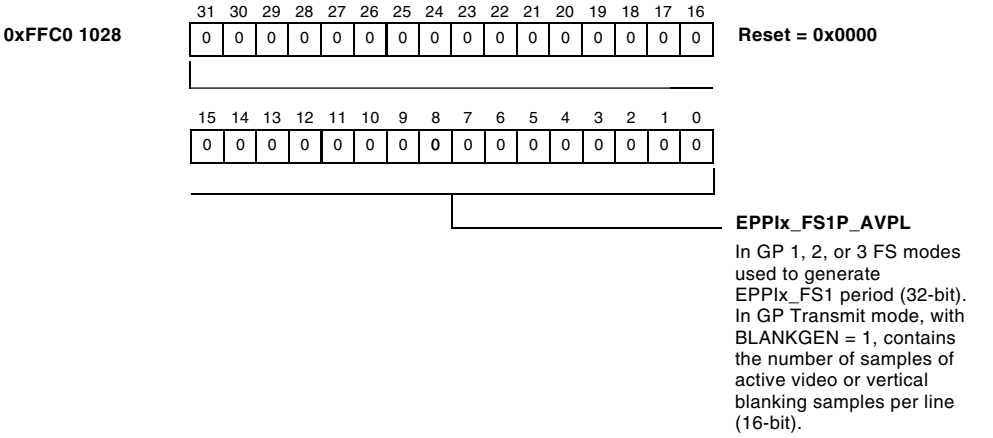


Figure 15-24. EPPI FS1 Period Register / EPPI Active Video Samples per Line Register

EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF)

The EPPIx_FS2_PERIOD register, shown in Figure 15-25, is a 32-bit register.

In GP 2 or 3 FS modes, it is used for the generation of Frame Sync 2. It contains the period required for FS2. The reference clock is EPPIx_CLK.

In GP Transmit mode with BLANKGEN = 1 in EPPIx_CONTROL, it contains the number of lines of active video per field.

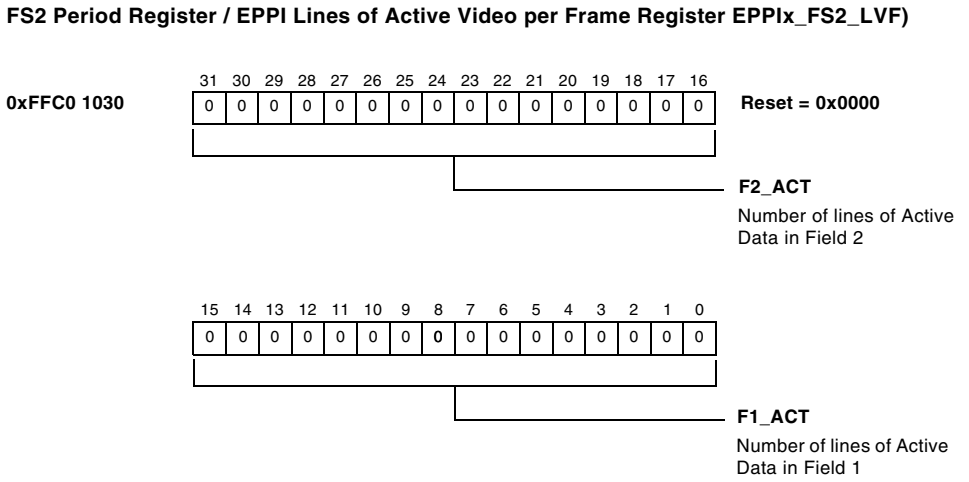


Figure 15-25. EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register

For progressive video, F2_ACT is ignored.

A value of 0 for F1_ACT or F2_ACT is illegal. If any of them is set to 0, the EPPI will regard its value as 1.

EPPI Clipping Register (EPPIx_CLIP)

The EPPIx_CLIP register, shown in Figure 15-26, is a 32-bit register used to define the lower and upper limits for the Luma and Chroma components. This is used for clipping of data values during 8-bit or 16-bit transmit modes. Refer to Figure 15-26 for bit definitions.

All data values for odd samples which are less than LOW_ODD are replaced with LOW_ODD and all data values for even samples which are less than LOW_EVEN are replaced with LOW_EVEN.

In the same manner, all data values for odd samples which are more than HIGH_ODD are replaced with HIGH_ODD and all data values for even samples which are more than HIGH_EVEN are replaced with HIGH_EVEN.

For 16-bit data lengths, the EPPI will separate each word into upper and lower bytes, and will consider the lower bytes as odd bytes and the upper bytes as even bytes during clipping.

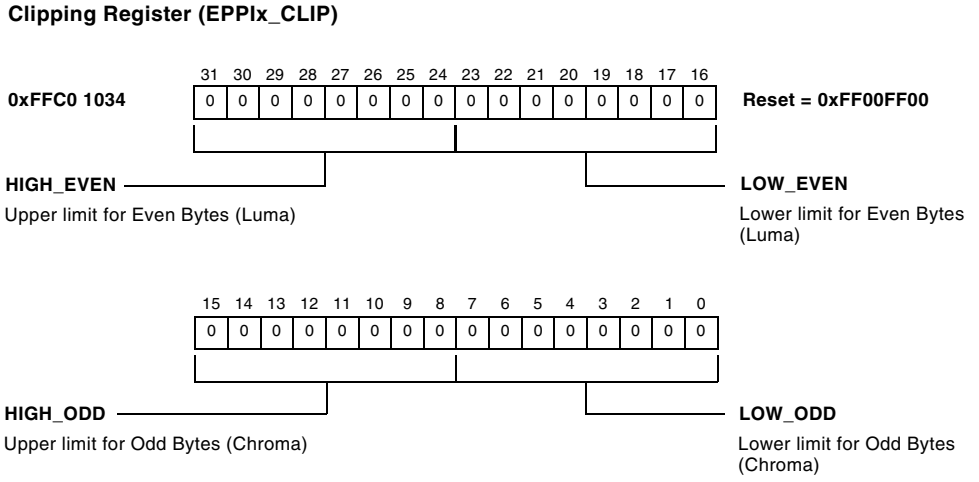


Figure 15-26. EPPI Clipping Register

In GP 0 FS mode with internal blanking generation, clipping is valid only for the active video part of the transmitted data. ITU-R 656 preambles, status words and blanking data bypass the clipping logic.

16 SECURITY

This chapter describes security features of the ADSP-BF54x processor Blackfin processor and how they can be used to facilitate a secure system.

This chapter includes the following sections:

- "Overview" on page 16-1
- "Description of Operation" on page 16-6
- "Programming Model" on page 16-32
- "Security Registers" on page 16-56

The intention of the chapter is to describe security features of the ADSP-BF54x processor Blackfin processor and how they can be used to facilitate a secure system. It is beyond the scope of this chapter to fully describe various ways to implement secure systems or to describe security protocols and primitives in any great detail.

Overview

Lockbox™ Secure Technology for Analog Devices Blackfin processors is comprised of a mix of hardware and software mechanisms designed to prevent unauthorized accesses and allow trusted code to execute on the processor. Throughout the rest of this chapter, the terms Blackfin Lockbox™ secure technology and Lockbox will be used interchangeably.

The developer's decision to use security features is completely optional. No security features are enabled by default. The developer can choose to never implement security features in their application if it is so desired. The Blackfin will always power up/boot in Open Mode with no security features or restrictions enabled.

Blackfin Lockbox™ secure technology allows users to:

- safeguard as little as a single function, a complete system, or anything in-between.
- uniquely identify each processor by a Unique Chip ID.
- utilize secure key storage provided by non-volatile, write-protectable One Time Programmable (OTP) memory.
- perform digital signature authentication using elliptic curve cryptography (ECC) and secure one-way hash (SHA-1) algorithms implemented in firmware.
- keep secret information in secure OTP Memory.
- use any encryption algorithm to protect code or other assets.
- ensure data integrity through digital signature authentication.
- safeguard confidentiality via encryption of any or all of the system -from core IP (code security) to data integrity.

These features in combination provide the following benefits.

- Authenticity/Origin verification—Lockbox secure technology allows for verification of a code image against its associated digital signature, and provides for a process to identify entities and data origins.
- Integrity—Developers can use a digital signature authentication process to ensure that the message or the content of the storage media has not been altered in any way. If either the message or digital signature was altered, Lockbox fails during the authentication process.
- Confidentiality—Cryptographic encryption/decryption supports situations that require the ability to prevent unauthorized users from seeing and using designated files and streams. Methods for ensuring confidentiality are supported by the secure processing environment (Secure Mode) and secure memory.
- Renewability—System components can be updated to enhance security.

  The Unique Chip ID enables end users to identify each Blackfin processor and hence each OEM device in which the processor resides.

  This Lockbox feature can be used in support of revocation and renewability of licenses in case of security violations in digital rights management systems. For example:

  Unique Chip ID—In combination with a trusted DRM agent (sourced by the OEM), this feature enables developers to implement renewability in DRM systems.

  Unique Chip ID—Provides capability to identify each OEM device and “blacklist” devices to remove them from a system.

- Prevention of mass copying—Lockbox supports cryptographic encryption/decryption algorithms for situations when confidentiality is required. The Unique Chip ID can also be utilized to “bind” the processor to one specific boot source/device and can be used to facilitate antitheft schemes and prevent OEM device cloning.

The ADSP-BF54x processor Blackfin processors featuring Lockbox™ secure technology provide security features that enable applications to use secure protocols consisting of code authentication and execution of code within a secure environment. Together these features protect secure memory spaces and restrict control of security features to authenticated developer code.

Features

Lockbox is comprised of a combination of hardware and software elements. These elements are:

- **OTP Memory**

  An array of non-volatile write-protectable memory that can be programmed by the developer only one time. Half of the array is public (accessible in any mode) and the other half is private (only accessible in secure mode). For more information on OTP memory, refer to Chapter 4, “One-Time Programmable Memory”.

- **Secured System Switches**

  Programmable bitfields in the Secured System Switches MMR to disable and enable different methods of memory access in support of a secured environment. Some of these protection mechanisms include disabling DMA access to L1 and L2 memory and disabling ADI JTAG instructions from the ICE port.

- **Secure Mode Control**

  This involves the Secure State Machine hardware required to support a transition from an unsecured state of operation (Open Mode), through an authentication state (Secure Entry Mode) and finally to a secured state (Secure Mode) where secrets are accessible.

- **Firmware**

  Code that resides in on-chip L1 instruction ROM and performs digital signature authentication. Having the code that performs the digital signature authentication in ROM ensures integrity of the code.

- **User callable cryptographic ciphers**

  In addition to the control code that resides in the on-chip L1 instruction ROM used for authentication, there exists a number of cryptographic functions (SHA-1, AES and ARC4) that are callable. The APIs are documented in "Programming Model" on page 16-32.

- **Unique Chip ID**

  Each ADSP-BF54x processor Blackfin processor has a 128-bit unique chip identification value stored in public OTP memory. The Unique Chip ID is programmed and write protected before a processor leaves the Analog Devices factory. It is always be located at the same OTP page address.

The 128-bit Unique Chip ID value can be read but cannot be modified by the developer or end user. A total of 64K-bits of OTP memory is available to the developer if additional user-defined ID values are desired. These IDs can be stored in either public or private areas of OTP memory depending on application requirements. Refer to Chapter 4, "One-Time Programmable Memory" for details.

Description of Operation

Blackfin Lockbox technology is based upon the concept of authentication of digital signatures using standards-based algorithms and provides a secure processing environment in which to execute code and access protected assets.

Digital signatures are created using a public-key signature algorithm, the Elliptic Curve Cryptography (ECC) public-key cipher, and a secure one-way hash algorithm, SHA-1. A public-key algorithm actually uses two different keys; the public key and the private key (called a key pair). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored in the public OTP memory region on-chip. Public-key algorithms such as ECC are designed so that if one key is used for encryption, the other is necessary for decryption. Furthermore, the encryption key cannot be reasonably calculated from the decryption key. In a digital signature authentication scheme like Lockbox, the private key is used to generate the signature and the corresponding public key is used to validate the signature. Each ADSP-BF54x processor Blackfin processor has an on-chip ROM that contains firmware with the Elliptic Curve Cryptography (ECC) and SHA-1 algorithms. These are called to verify the digital signatures (ECDSA[1]).

JTAG emulation and test features are disabled in hardware and certain memory access restrictions are enabled during verification of the digital signature. Once the signature is authenticated, the access restrictions are still in effect and can only be controlled by the authenticated user code.

[1] ECDSA implementation on ADSP-BF54x processor Blackfin products only supports the Koblitz curve.

Secure State Machine

The ADSP-BF54x processor processor includes a Secure State Machine to handle the different protection configurations of the processor depending on the security situation. The machine states are “Open Mode”, “Secure Entry Mode” and “Secure Mode” (See Figure 16-1). The following sections describe these machine states.

The state of the Secure State Machine can be identified by reading bits in the SECURE_STATUS[1:0] register. The bit values in the upper right of the states shown in Figure 16-1 correspond to the bit values in SECURE_STATUS[1:0].

For more information on the SECURE_STATUS register, see “Security Registers” on page 16-56.

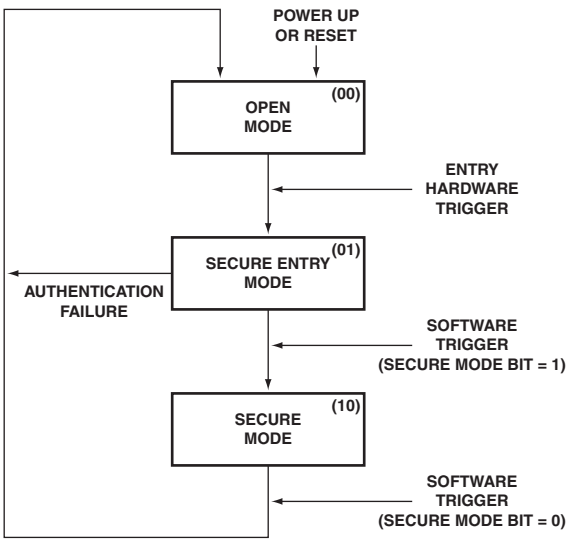


Figure 16-1. Secure State Machine Modes

Open Mode

This is the default operating state of the processor, in which no restrictions are present except restricted access to the Private OTP memory area. The processor powers up and boots in Open Mode. This is the default state upon power up and after processor reset. No Lockbox security features or protection mechanisms are enabled in this state.

The state flow illustrated in Figure 16-1 shows that the Secure State Machine can only transition from Open Mode into Secure Entry Mode, and there is no direct path from Open Mode into Secure Mode.

Secure Entry Mode

The on-chip ROM firmware performs the authentication process in this operating state. This mode is entered when NMI is active and the program counter (PC) is vectored to the first address of the authentication firmware in the on-chip ROM. The program counter is monitored to ensure that it remains within the address range allocated to the Authentication firmware code. If the program counter vectors outside of the address range of the authorization code, authentication fails and the state returns to Open Mode. Any errors caught by firmware or hardware monitor will result in authentication failure and an abortion of the authentication process with the firmware exiting Secure Entry Mode and transitioning back to Open Mode. If authentication is successful, the firmware initiates the transition from Secure Entry Mode to Secure Mode.

In Secure Entry Mode, no DMA access is allowed to certain regions of internal SRAM, and JTAG emulation is disabled. The user should disable cache prior to initiating authentication. Interrupts are disabled by firmware prior to entry into Secure Mode. Interrupts are either re-enabled by dropping the interrupt level from NMI via the SESR arguments or by waiting until the authentication is successful and re-enabling them in the authenticated code after entry into Secure Mode. In addition, only the

public area of OTP memory is accessible in this mode. For more information on memory access restrictions within Secure Entry Mode, see "Secure Entry Service Routine (SESR) API" on page 16-32.

State flow, illustrated in Figure 16-1, shows that the Secure State Machine can only transition from Secure Entry Mode to Secure Mode upon successful digital signature authentication. A transition from Secure Entry Mode back into Open Mode can occur if digital signature authentication fails or if the authentication process is aborted due to an error observed by the firmware. Such errors include illegal memory boundary conditions or jumps outside of the firmware range (for example, servicing an interrupt).

Secure Mode

In the secure operating state trusted, authenticated code is allowed unrestricted access to the processor resources, execution of authenticated code occurs, decryption of sensitive information, etc. This is the only mode that allows access (reads and writes) to the private OTP memory space—where secure data such as secret keys can be stored. The private area of OTP memory can be used to store confidential, secret information that only authorized authenticated code can access. This is the only operating state where users can securely run their own Blackfin implementation of any cryptographic cipher in which secret keys are used.

Only the code (or message) digitally signed by a trusted source and successfully passes through Lockbox's authentication process can gain access to Secure Mode.

The state flow illustrated in Figure 16-1 shows that the Secure State Machine can only transition from Secure Mode back into Open Mode and there is no direct path from Secure Mode into Secure Entry Mode. Exit from Secure Mode is implemented through software control by writing a "0" value to the `SECURE0` bit within the `SECURE_CONTROL` register.

Assertion of reset or power cycling will also return the processor to the default Open Mode regardless of the state of operation when the reset or power cycle event occurred. See special handling of hardware reset in "Reset Handling in Secure Mode" on page 16-21.

Access to private OTP memory is restricted in Open Mode and Secure Entry Mode regardless of whether or not other security features are enabled or disabled.

SecureMode Control

Figure 15-2 describes the inputs that control the secure state machine flow.

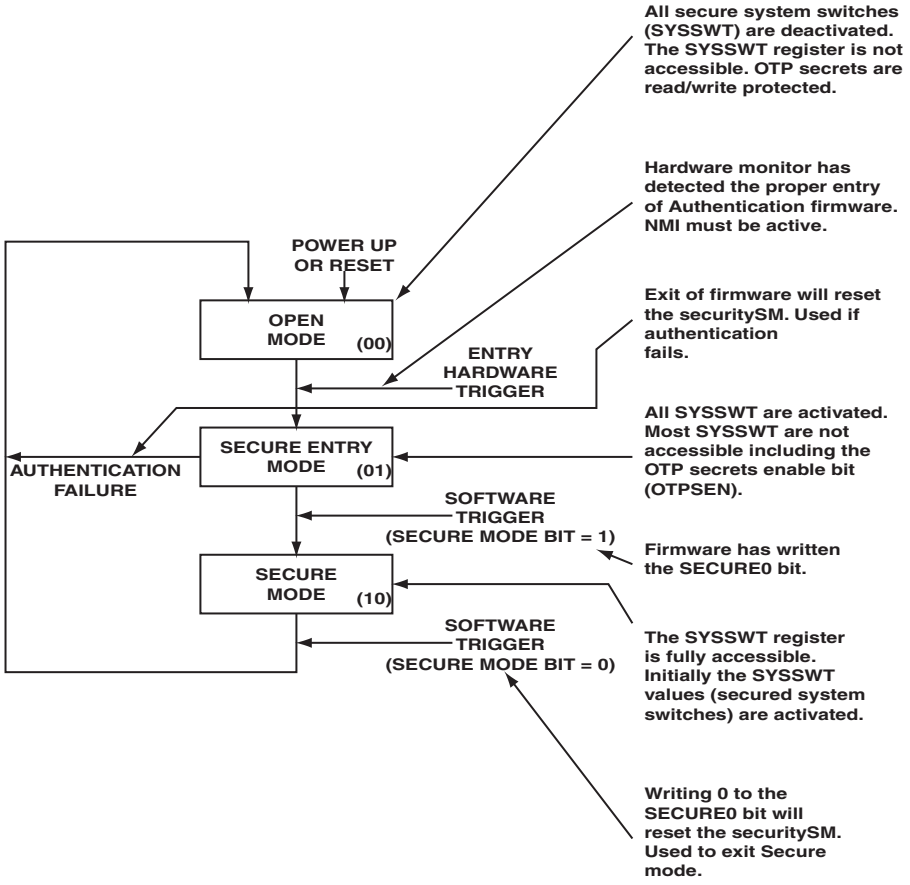


Figure 16-2. Secure Mode Control

Hardware supports transition from an Open Mode of operation, through a Secure Entry Mode to a Secured Mode where secrets are accessible.

Open Mode is characterized by being the default mode of processor upon power up/reset/boot, holding all secured system switches deactivated and protecting the private OTP memory area from access. The processor is open with all features being available with no restrictions (except for the private area of OTP memory).

Secure Entry Mode is characterized by executing firmware out of internal ROM memory to authenticate information loaded into on-chip memory. All secured system switches are activated. However, private OTP Memory is not accessible yet.

Secure Mode is entered only after a successful digital signature authentication process from Secure Entry Mode. It provides access to the private OTP memory area and makes secured system switches accessible to user (authenticated) code. This is the mode of operation in which to perform sensitive decryption or execution of trusted, authenticated code.

Authentication can only be requested and initiated while the processor is operating in Open Mode. If authentication is requested while the processor is operating in Secure Mode, the Secure State Machine will not transition into Secure Entry Mode. Instead, the Secure State Machine will remain in Secure Mode.

Open Mode, Secure Entry Mode and Secure Mode are states which pertain to the Secure State Machine. User Mode and Supervisor Mode are modes of operation which pertain to the core. The use of the term “mode” should not be confused and are not necessarily mutually exclusive. In Open Mode, the processor can operate in either User or Supervisor Mode. Since the firmware is entered when the NMI is being handled, Secure Entry Mode must start in Supervisor Mode. Finally, authenticated code executing in Secure Mode must be either operating at NMI interrupt level or the interrupt level that triggered the NMI.

Security Features

The following sections provide a functional description of the Security features.

Protection relies on the on-chip ROM code that includes Elliptic Curve Cryptography (ECC) and SHA-1 algorithms, applied towards verification of code authenticity using a digital signature. A processor has emulation and test features disabled in hardware as well as certain memory access restrictions upon entry into Secure Entry Mode (where authentication is performed) and maintained into Secure Mode. These functions can be controlled only by authenticated user application software executing in Secure Mode.

User code must request authentication by complying with two criteria: (1) asserting a Non-Maskable Interrupt (NMI) and (2) vector the program counter (PC) to the first executable address in the Secure Entry Service Routine (SESR) in firmware which resides in L1 Instruction ROM.

During the authentication process, JTAG emulation is disabled, memory protection restrictions are enabled and interrupts are masked. The user has the option to pass arguments to the security firmware to control certain functionality during the authentication process. Refer to "Secure Entry Service Routine (SESR) API" on page 16-32.

Digital Signature Authentication

Digital signatures are created off-chip (typically on a host computer) using the ECC algorithm and SHA-1, both of which are available in the public domain. In digital signature authentication, the private key generates the signature (off-chip) and the corresponding public key validates the signature (on-chip). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored on-chip in OTP memory.

Lockbox uses standards-based cryptographic algorithms for digital signature authentication. ECDSA[1] is implemented in the Blackfin ADSP-BF54x processor processors. Digital signature validation on

[1] ECDSA implementation on these Blackfin products only supports the Koblitz curve.

ADSP-BF54x processor utilizes Elliptic Curve Cryptography[1] (ECC) based on a binary field size of 163 bits and SHA-1[2] secure one-way hash (which produces a 160-bit message digest).

In order to generate public/private key pairs or prepare digital signatures and apply them to application code, developers can use any method that complies with the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS). ECDSA is described in ANSI X9.62-1998. The Lockbox implementation in the ADSP-BF54x processor processors supports the following Koblitz curve, which is recommended in FIPS 186-2 for US Federal Government use:

m: 163 (degree of binary field)

a: 1

b: 1 (a and b are the constants in the elliptic curve equation: $y^2 + xy = x^3 + ax + b$)

X_g: 2FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8

Y_g: 289070FB05D38FF58321F2E800536D538CCDAA3D9 (X_g and Y_g define the base point G)

r: 4000000000000000000020108A2E0CC0D99F8A5EF (r is the order of the base point G)

T: 4 (T is the normal basis type)

p(t): $t^{163} + t^7 + t^6 + t^3 + 1$ (pt(t) is the field polynomial)

The following steps summarize the Digital Signature Authentication process. Steps 1 to 3 correspond to the off-chip creation of a digital signature of a file or message. Steps 4 to 6 correspond to the on-chip digital signature authentication. These steps are preceded by generation of a key pair (Private Key and Public Key) and the programming of the Public Key in the Public OTP Memory.

---

[1] These implementations are based on the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS) (http://csrc.nist.gov/cryptval/dss.htm), and specified in ANSI X9.62-1998.

[2] SHA-1 is based on the publicly available standard for FIPS 180-2 (Secure Hash Signature Standard [SHS]) (FIPS PUB 180-2), http://csrc.nist.gov/CryptoToolkit/tkhash.html).

1. A one-way hash of the file (message to be authenticated) is produced using SHA-1 off-chip (for example, using a host PC).
2. The hash is encrypted through ECC off-chip with the private key, thereby signing the file and completing the generation of the digital signature.
3. The file and the signed hash are stored on an external device such as Flash memory or a host device.
4. Upon transfer to the Blackfin processor's internal memory, a one-way hash of the file is calculated on-chip through SHA-1 (residing in Blackfin on-chip ROM).
5. Using the ECC algorithm (residing in the Blackfin on-chip boot ROM), the Blackfin decrypts the signed hash with the user's public key stored in the Blackfin's OTP memory.
6. The two hash results are then compared. If the signed hash matches the calculated hash, the signature is valid and the file is intact.

If the digital signature authentication process is successful, the Blackfin processor will transition from Secure Entry Mode to Secure Mode. At this time, all of the access restrictions mentioned will be in place. JTAG will be disabled and certain portions of on-chip SRAM memory are restricted from DMA access. The restrictions can be controlled once in Secure Mode by having the authenticated code modify the Secure System Switches (`SECURE_SYSSWT`) appropriate for use by the developer's application.

Encryption/decryption is only necessary when an application requires *confidentiality*. It is not always necessary to work with encrypted code to ensure code security. Authentication alone can be used when confidentiality is not required when ensuring tamper-proof code image and/or non-repudiation in a system. Authentication thus safeguards code *integrity*.

Since the digital signature uniquely describes its corresponding code/message, the code/message itself does not have to be encrypted if *confidentiality* is not required. If the code/message is modified, either intentionally or inadvertently, authentication fails since the *integrity* of the code message has been compromised.

Digital Signature Authentication Performance Measurement

Authentication can be performed at any point during processor operation in Open Mode. It can be performed immediately upon boot or it can be performed any time after boot.

The algorithms used in the Lockbox firmware are highly optimized Blackfin code running from L1 instruction ROM in the core clock domain. Firmware execution time for the digital signature authentication process is on the order of 5 million core clock cycles depending upon the size of the digitally signed application code. This must be considered when architecting an application in order to allow a sufficient window of time in which authentication can proceed without requiring servicing of interrupts in the system.

The time it takes for authentication is dependent on several factors. These include the size of the message to be authenticated. This affects the amount of calculations done in the secure hash function (SHA-1). It also affects the DMA time required to move the message out of L1 data memory and place it into L1 code memory.

Protection Features

In order to establish a secure processing environment and protect the security of applications that establish trust and reach the privileged mode of operation, Lockbox implements access restrictions. These restrictions include disabling JTAG emulation and disabling DMA access to portions of on-chip SRAM memory. The memory access restrictions implemented in hardware on the Blackfin processor are not applied to off-chip memory. Therefore, external memory is always considered insecure and caching external memory while operating in Secure Mode represents a security risk.

Protection features include the following:

- Secure State Machine for implementing privileged states of operation in which access restrictions may be imposed to protect code and data.
- Disable DMA access to L1 and on-chip L2 memory
- These restrictions to memory areas are configurable (see "Secured System Switches (SECURE_SYSSWT) Register" on page 16-57).
- Protection of L1 and on-chip L2 regions of memory with DMA access controlled when in Secure Mode.
- Disable ADI JTAG emulation from ICE port
- Divert hardware reset to NMI during Secure Mode operation to prevent "reset attack".
- Provide software control over hardware protection features accessible to trusted code operating in Secure Mode.
- OTP memory for storage of customer programmable cipher keys, unique chip ID or a customer ID

- OTP write protection to protect programmed OTP memory locations from future tampering
- Private/Secret OTP memory region accessible only in Secure Mode
- Store private key(s) for decryption of data or other validation
- A privileged mode (including firmware execution out of on-chip ROM) to perform code authentication

Protection mechanisms are summarized Table 16-1 for each state of the Secure State Machine along with the Secure System Switch register (SECURE_SYSSWT) that provides control over the protection feature.

Table 16-1. Secure State Machine

Secure State Machine	SECURE_SYSSWT	Description	Protected Memory Range
Open Mode (0x0000 0000)	The switches are involuntarily set with all controls OFF (unrestricted access).	No protection mechanisms or restrictions enabled.	No restrictions[1]
Secure Entry (0x0007 04D9)	EMUDABL	Emulation Disable	Emulation disabled
	L1IDABLE	L1 Instruction Memory Disable 0xFFA0 0000 - 0xFFA0 7FFF SRAM	32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF80 0000 - 0xFF80 7FFF SRAM and SRAM/Cache	32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF90 0000 - 0xFF90 1FFF SRAM	8 KB
	L2DABL	l2 Memory Disable	64 KB
Secure Mode (0x0007 04D9)	EMUDABL	Emulation Disable	User Configurable
	RSTDABL	RESET Disable	User Configurable
	L1IDABLE	L1 Instruction Memory Disable 0xFFA0 0000 - 0xFFA0 7FFF SRAM	0-32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF80 0000 - 0xFF80 7FFF SRAM and SRAM/Cache	0-32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF90 0000 - 0xFF90 1FFF SRAM	0-32 KB
	L2DABL	l2 Memory Disable	0-64 KB

1 Private OTP is only accessible when operating in Secure Mode with OTPSEN bit set in SECURE_SYSSWT register

On-chip SRAM memory protection takes the form of DMA access restrictions only. There is no need to protect the on-chip SRAM from processor core access because, while operating in Secure Mode, the developer's authenticated code has full control over the processor core and execution of all core software instructions. It is the responsibility of the developer to take steps to avoid surrendering control of the Program Sequencer and the core to untrusted code execution.

Operating in Secure Mode

The following sections describe how to enter and exit secure mode.

Entering Secure Mode

Upon successful digital signature authentication, the Secure State Machine transitions into Secure Mode. The same default protection features enabled in Secure Entry Mode are carried forward into Secure Mode. This includes JTAG emulation being disabled, and DMA access restrictions to memory and interrupts being masked. It is the responsibility of the authenticated code to manipulate or remove these restrictions as desired.

Exiting Secure Mode

Secure Mode provides a secure operating environment to execute sensitive code, run cryptographic ciphers and process sensitive data. Upon exiting Secure Mode, the authenticated code should remove any sensitive code and data from memory because this sensitive information will still be accessible in Open Mode if it is not removed prior to exiting Secure Mode. Exit from Secure Mode is implemented through software control by writing a "0" value to the SECUREO bit within the SECURE_CONTROL register. Refer to "Security Registers" on page 16-56 and "Clearing Private Data" on page 16-22 for more information.

Reset Handling in Secure Mode

The following describes how hardware is reset, and how to clear private data/

Hardware Reset

Hardware reset is diverted to NMI when operating in Secure Mode only. When operating outside of Secure Mode, hardware reset behaves normally. This protection feature is configurable via the RSTDABL bit within the SECURE_SYSSWT register when operating within Secure Mode.

This is a protection feature to prevent malicious entities from attempting to assert hardware reset while sensitive code or data is present in the processor's on-chip SRAM or in the processor's registers. A "reset attack" could take the following form: If hardware reset were left unprotected and reset was asserted while sensitive information were present on-chip, the processor would return to the default state of Open Mode with no protection features enabled and a malicious entity could gain access to the on-chip memory and registers, for example via JTAG emulation. In such a scenario assets intended to be protected could be compromised.

By diverting hardware reset to NMI while the processor operates in Secure Mode, servicing of hardware reset can be controlled and delayed in order to first implement a memory clean-up routine in software to purge sensitive information from internal memory and registers prior to servicing reset. At the completion of the memory clean-up, the processor can then be reset via software command and safely returned to Open Mode with no sensitive information available to be compromised.

By default, the SESR loads the address of a memory clean-up routine stored in the on-chip L1 instruction ROM into the NMI EVT2 prior to transitioning from Secure Entry Mode into Secure Mode. See "Clearing Private Data" on page 16-22 for more information.

Clearing Private Data

As part of the SESR firmware, there is a small routine stored in the on-chip L1 instruction ROM that clears the internal memory, generates a RESET event and puts the processor into idle. It is recommended that the user sets this routine as the new EVT2 NMI vector once the user's authenticated application code is executing. This will prevent a malicious user from trying to reset the processor while it is operating in Secure Mode and then view the contents of internal memory when the processor returns to Open Mode after servicing RESET. The "Clear Private Data" routine is located at address 0xFFA1 47E8.

It is recommended that user software running in Secure Mode should also perform RAM clean-up prior to clearing the SECURE0 Secure Mode bit and exiting Secure Mode via normal code execution within user's secure function. If sensitive code/data remains in on-chip RAM after exiting Secure Mode without wiping memory and register contents or cycling power to the processor, it will be visible and accessible in Open Mode.

This memory wipe routine in the ROM executes a watchdog RESET to reset the processor at the completion of the memory wipe. The code also performs a wipe of the OTP_DATA0-3 registers which are used to hold data from OTP access reads (that is, which could contain secret key or other sensitive data left by user code execution).

If a custom memory cleanup routine is part of an authenticated message, the user can use that routine instead of the one provided with the Lockbox firmware. The user just has to update EVT2 in the event vector table to point to the start of the custom memory cleanup routine.

Due to the fact that hardware reset is configured by default to be redirected to NMI when the processor is operating in Secure Mode, it is recommended that the user implements a watchdog reset within the EVT2 NMI ISR in order to reset the processor. A Watchdog reset is implemented by writing a value 2'b00 in WDOG_CTL[2:1] which causes reset of

both the core and the peripherals, excluding the RTC block and most of the DPMC. The watchdog reset will not be redirected to the NMI pin as in the case of the external hardware reset and it will properly reset the processor. For more details of watchdog reset, refer to "Software Resets" in Chapter 17, System Reset and Booting.

This "reset attack" protection scheme needs to protect only against hardware reset which can be applied externally as the system developer typically has no control over reset in an embedded system. While operating in Secure Mode, the developer's authenticated code has full control over the processor core and execution of all software instructions, so there is no need to protect against soft reset instructions. It is not recommended that the user's secure application code implement a soft reset without first deleting sensitive information from memory and registers.

Public Key Requirements

A valid ECC public key must be a non-zero value and meet the following criteria:

Given the public key value shown here:

3693 68AF 2431 93D0 01E3 9CE7 6BB1 D5DA 08A9 BC0A 615F 7A90 C841 D4F1 E1B0 05E7 0F16 7F6E F7CD 2E25 1B

format in 32-bit little endian as follows:

8A9B C0A6

BB1D 5DA0

1E39 CE76

4319 3D00

6936 8AF2

0000 0003

CD2E 251B

167F 6EF7

B005 E70F

41D4 F1E1

5F7A 90C8

0000 0001

The values should be stored in OTP pages 0x10, 0x11, 0x12 as follows (where 'L' denotes lower half of page, 'H' denotes upper or high half of page):

page: 0x010L: 0xBB1d 5DA0 8A9B C0A6

page: 0x010H: 0x4319 3D00 1E39 CE76

page: 0x011L: 0x0000 0003 6936 8AF2

page: 0x011H: 0x167F 6EF7 CD2E 251B

page: 0x012L: 0x41D4 F1E1 B005 E70F

page: 0x012H: 0x0000 0001 5F7A 90C8

The general format takes the form of twelve (12) 32-bit words:

Word 1

Word 2

Word 3

Word 4

Word 5

Word 6

Word 7

Word 8

Word 9

Word 10

Word 11

Word 12

Stored into OTP pages in the following order (where 'L' denotes lower half of page, 'H' denotes upper or high half of page):

page: 0x010L: Word 2 Word 1

page: 0x010H: Word 4 Word 3

page: 0x011L: Word 6 Word 5

page: 0x011H: Word 8 Word 7

page: 0x012L: Word 10 Word 9

page: 0x012H: Word 12 Word 11

Storing Public Cipher Key in Public OTP

In order to make use of security features, the user must first store an ECC public key in the Blackfin processor public region of OTP memory pages 0x10, 0x11 and 0x12 as specified in the Firmware's Secure Entry Service Routine (SESR) API and the OTP memory map (see "Secure Entry Service Routine (SESR) API" on page 16-32). If no ECC public key is stored in this area of OTP, digital signature authentication cannot be successfully completed and no Lockbox security features can be enabled. For more information see Chapter 4, "One-Time Programmable Memory".

If security features which rely upon the ECC public key are not going to be used, it is recommended that customers write-protect the ECC public key OTP memory space in order to prevent malicious entities from writing a value into this memory and potentially exploiting this feature without the developer's consent.

Cryptographic Ciphers

Lockbox uses SHA-1 and ECC to implement ECDSA as part of the authentication process to enter into Secure Mode. These ciphers reside in the firmware in the on-chip L1 instruction ROM. In addition to these ciphers, the Advanced Encryption Standard (AES) and ARC4 are also available in the ROM. The SHA-1, AES and ARC4 ciphers are user-callable in Open Mode or in Secure Mode. The APIs are documented in "Programming Model" on page 16-32r. Note that ECC is not user-callable and is only executed as part of firmware during the authentication process.

Since AES uses symmetric keys that need to be private, and these private keys typically require confidentiality, it is recommended that this cipher be executed in Secure Mode to access the keys from the private area of OTP memory.

Keys

Although Lockbox uses an ECC public key for digital signature authentication, and has private OTP memory to store private keys for other cryptographic algorithms, Lockbox does not implement key management. Lockbox does not implement key generation nor does it implement key exchanges natively in the Blackfin hardware.

In order to use Lockbox, an ECDSA key pair must be generated. The private key is used off-chip (typically on a host PC) to sign the message. The public key is placed in the public OTP memory where it is used to authenticate the signed message. Lockbox is only part of a full cryptosystem. It is the responsibility of the user to develop the other parts of the cryptosystem necessary for the intended application.

Debug Functionality

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard. 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.

ADSP-BF54x processor debug functionality has some modified behavior dependent upon the access privileges associated with the state of the Secure State Machine operating mode. This is to ensure that sensitive information and processing performed within Secure Entry Mode and Secure Mode will not be compromised via JTAG. Furthermore, public JTAG instructions necessary for system test and debug (such as boundary scan and bypass mode) remain in effect regardless of the state of the Secure State Machine and are not hindered by ADSP-BF54x processor Secure Mode operation. This makes it possible for developers to debug their systems without interference from the Blackfin processor or its security features.

In compliance with the JTAG standard, ADSP-BF54x processor processors provide an Instruction Register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation. 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.

All supported public and private JTAG instructions remain operational when operating in Open Mode. All supported public JTAG features remain operational and all private JTAG features are disabled when operating in Secure Entry Mode and Secure Mode. Refer to Appendix B, "Test Features" for more information about supported JTAG instructions.

By default, JTAG emulation is disabled when the processor enters Secure Entry Mode or Secure Mode. There is only one way to enter Secure Mode—through successful authentication of user code based on digital signature validation. Once the digital signature authentication process results in success, the user's trusted, authenticated code is given full control over the processor, including access to Secured System Switches register (SECURE_SYSSWT) that enables/disables various protection mechanisms, including JTAG emulation. The Secured System Switch register provides a setting that will allow authenticated code to enable JTAG emulation either in a one-time secure session setting or in a 'sticky' persistent manner that allows emulation to be enabled by default the next time the processor enters Secure Mode. These settings are cleared when reset is asserted or if processor core power is cycled. See the EMUOVR and EMUDABL bits in the SECURE_SYSSWT Secure System Switches Register in "Secure System Switch Register, Bits 15:0" on page 16-58.

Two bits within the SECURE_SYSSWT Secure System Switches register control JTAG emulation; they are Emulation Disable (EMUDABL) and Emulation Override (EMUOVR). To enable JTAG emulation for the current session while operating within Secure Mode, SECURE_SYSSWT bit 0 (EMUDABL) must be set to 0. To enable JTAG emulation to remain 'sticky' and persistently enabled for the current session and for all subsequent entries into Secure Mode until cleared by the user or until cleared via RESET or cycling power to the processor, SECURE_SYSSWT bit 0 (EMUDABL) must be set to 0 *and* SECURE_SYSSWT bit 14 (EMUOVR) must be set to 1 simultaneously. See "Secure System Switch Register, Bits 15:0" on page 16-58 for details.

The EMUDABL bit is only directly writable when in Secure Mode. EMUOVR can be written to a 0 at any time. RESET will clear EMUOVR. EMUOVR can be cleared by the user at any time and in any mode, including Open Mode, Secure Entry Mode, and Secure Mode. You do not have to operate in Secure Mode in order to clear EMUOVR.

The EMUDABL bit is only directly writable when in Secure Mode. EMUOVR can be written to a 0 at any time. This means if you are in Secure Mode and wish to remove the privilege of emulation override, you are allowed to clear EMUOVR. Or if you are operating in Open Mode and wish to remove emulation override, you can clear EMUOVR. In the case of Secure Entry Mode, writing the EMUOVR bit to a 0 immediately blocks emulation (and the EMUDABL bit would read 0 immediately. While Operating in Secure Entry Mode, the value of EMUDABL is the *not* of EMUOVR, that is, EMUDABL = ~EMUOVR. While operating in Secure Mode, you can read or write the EMUOVR bit which has no immediate affect since EMUDABL is in control at that point.

Upon setting EMUDABL = 0 *and* EMUOVR = 1, JTAG emulation remains active and enabled for the current session during Secure Mode operation *and* for *all* subsequent entries into Secure Mode until EMUOVR is cleared (set to 0) or until RESET or power cycle clears this setting. This is also known as "sticky" emulation setting.

If 'sticky' emulation is enabled (EMUDABL = 0 *and* EMUOVR = 1), JTAG emulation will be active and enabled in all modes, that is, Secure Entry, Secure Mode as well as Open Mode. The Secure State Machine can cycle through all modes of operation, and JTAG emulation will remain active and enabled in every mode with these settings in place until cleared by the user application code, or until RESET or power cycle clears the setting.

For example, a user creates code to be authenticated with a valid digital signature. The code and digital signature are loaded onto the Blackfin processor in Open Mode, Authentication is requested (JTAG emulation is disabled by default during Authentication in Secure Entry Mode) and the Authentication process is successful. The processor enters Secure Mode

(JTAG emulation still disabled by default) and control is given to the authenticated code. Authenticated code sets bits within the Secure System Switches to enable JTAG Emulation and sets the 'sticky' bit to allow JTAG emulation to be enabled by default the next time the processor transitions into Secure Mode as well. Debug within Secure Mode can occur using emulation now. If a different set of trusted code must be loaded into the processor, the user can do so now without leaving Secure Mode or the user can choose to exit Secure Mode and return back to Open Mode in order to authenticate another set of code or load test/problematic code. A new set of code and digital signature can now be loaded and authenticated. Upon entry into Secure Mode. JTAG emulation will be enabled by default due to the sticky bit setting in the Secure System Switches. Debug can be performed within Secure Mode without changes to problematic code.

One possible usage scenario for persistent (sticky) emulation might be as follows: a “final” production code that must run in Secure Mode is prepared. There seems to be an issue with the code but emulation prevents working with it. You would take advantage of EMUOVR bit within the SECURE_SYSSWT register by first performing a simple authentication of code that sets the EMUOVR bit in order to enable JTAG emulation within Secure Mode. From there you exit Secure Mode (write a value of "1" to the SECURE0 bit in the SECURE_CONTROL register, but do not invoke any processor reset), and call the routine to debug. You would then set a breakpoint just after authentication. That way you could now step through your code using JTAG emulation, and operate in Secure Mode.

Digitally signed user code, which enables either single session or sticky JTAG emulation, must be treated as confidential by users in the same manner as private keys. If this code is allowed to fall outside of developer control or become public, it can be used to compromise a developer’s security.

In summation, to enable JTAG emulation during Secure Mode, the user must successfully perform the Authentication process at least one time, and then program the Secured System Switches while operating in Secure Mode to enable emulation.

Programming Examples

Listing 16-1. Assembly Code – Enable ("Sticky") Persistent JTAG Emulation for Secure Mode Debug

```
  #include <defBF548.h>
  .section L1_code;
  .align 4;
  .global _secure_function;
  _secure_function:
  /* required nops to account for SESR PC vector target+4 for
overlay ID accommodation*/
  nop;
  nop;
  P0.H = ((SECURE_SYSSWT) >> 16);
  P0.L = ((SECURE_SYSSWT) & 0xFFFF);
  R0 = [P0];
  BITCLR(R0,0);
  [P0] = R0;
  SSYNC;
  _secure_function
.END:
```

Listing 16-2. C Code – Enable JTAG Emulation for Secure Mode Debug (single session)

```
  #include <cdefBF548.h>
  #define    ENABLE_JTAG_MASK       0xFFFFFFFE
  void secure_function(void)
```

```
{
  /* Enable JTAG */
  *pSECURE_SYSSWT = ( *pSECURE_SYSSWT & ENABLE_JTAG_MASK );
  ssync();
  return;
}
```

Programming Model

The following sections describe the programming model for security features.

Secure Entry Service Routine (SESR) API

This section describes the procedure to use Lockbox to authenticate a message. Memory configuration, input arguments and return codes are also described here.

In this chapter, the term “message” was widely used to describe the entity being digitally signed off-chip, and later authenticated on-chip by the SESR security firmware. “Message”, “secure function” (SF), and “secure application” are used interchangeably in this section and mean the same thing.

Starting Authentication

For an application to establish trust and reach the privileged mode of operation (for example, enter Secure Mode), the Secure State Machine has to transition from Open Mode, through Secure Entry Mode, to Secure Mode. In order to transition from Open Mode to Secure Entry Mode, NMI must be asserted and the program counter (PC) must vector to the beginning address of the firmware (SESR).

This can be achieved by loading `BFROM_SECURE_ENTRY` (defined in bfrom.h) as the NMI handler in the event vector table (EVT2). Then in supervisor mode, issue a `raise 2` instruction. Similarly, the NMI hardware pin may be asserted instead of issuing a software raise instruction. Once the program counter vectors to the SESR, and while NMI assertion is sensed by the hardware, the Secure State Machine will transition into Secure Entry Mode.

Before actually going into Secure Entry Mode, the user will have to set up the memory environment. This includes specifying the arguments (described in this section) and moving the message to be authenticated into L1 data memory.

Memory Configuration

Figure 16-3 illustrates the Secure Entry Mode default memory configuration upon initiating authentication and entering the SESR.

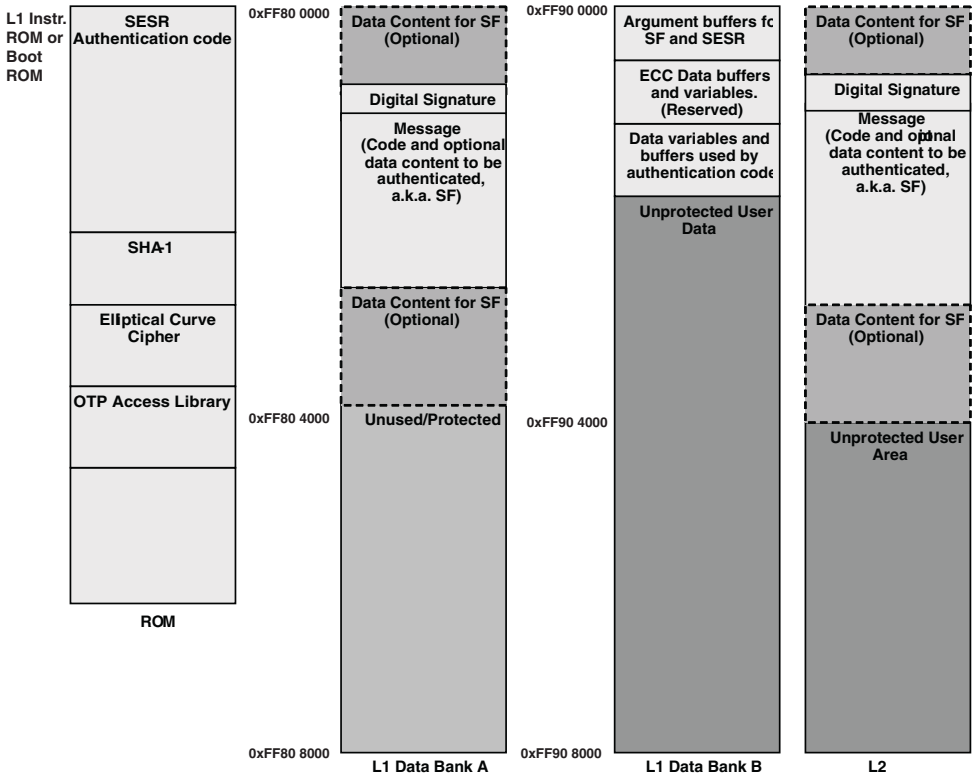


Figure 16-3. Memory Configuration for Authentication

Message Placement

The message can be placed in either L1A or in L2 for authentication. If the message (for example, code) is put into L1A for authentication, it must be DMA'ed to either L1 code space or L2, where it can execute. If the message is placed into L2 for processing, it has the option of staying

where it is and can be executed directly from L2. It is the user's responsibility to provide the message in L1A or L2 memory for the SESR. If authentication is successful, the SESR will then move the message via DMA to the final destination according to the SESR arguments. No further action is required by the developer to perform this DMA as it is executed by the firmware.

Digital Signature

The digital signature is a pair of 163-bit integers. Each integer is padded to the nearest 32-bit word, resulting in 192 bits for each integer resulting in a total size of 384 bits. The authentication firmware always expects the digital signature to be followed by the message. For example, if the message is placed in L1A data memory, and the digital signature starts at address 0xFF80 0000, the message must immediately follow the digital signature and be located at address 0xFF80 0030. The same holds true if the message and digital signature are placed in L2 memory as they must be stored together contiguously in memory with the message always immediately following the digital signature.

Message Size Constraints

The maximum size of any message to be authenticated is limited by the size of on-chip memory in the Blackfin. When the Secure State Machine enters into Secure Entry Mode (authentication), certain portions of on-chip SRAM memory are protected from DMA accesses. These protected memory regions include L1A (32 KB) and L1B data memory (8 KB each), L1 code memory (32 KB) and half of L2 memory (64 KB). This means that the maximum allowable message/code size that can be authenticated is 32 KB less 48 bytes for the digital signature, if placed in L1A data memory and 64 KB less 48 bytes if placed in L2 memory.

Memory Usage

In data bank B of the L1 memory, the arguments for both the SESR and the secure function are stored beginning at address 0xFF90 0000. In addition, a portion of the L1B data memory is reserved for the firmware for scratch space. All memory above address 0xFF90 1F00 is reserved for authentication. The user can either allocate this area of memory solely for Lockbox or save any data elsewhere in memory prior to starting authentication.

Any user information residing in the scratch space reserved area of L1 Data Bank B will be overwritten during the authentication process.

Memory Protection

This Secure Entry Mode default memory configuration with both protected and unprotected regions of on-chip SRAM is implemented in order to allow developers to initiate digital signature authentication at any time during Open Mode processor operation. If an application is already running on the processor, the unprotected memory regions can be used for placement of data buffers. When authentication occurs, access to these data buffers is not restricted thus the application can be given higher precedence over the authentication process if necessary.

The Secure Entry Mode default memory protection configuration put into place upon initiating authentication cannot be modified by the developer. This is to ensure integrity of the secure processing environment during the authentication process and help prevent malicious tampering.

Secure Function and Secure Entry Service Routine Arguments

Prior to initiating the authentication, the arguments for both the SESR and the message (also known as Secure Function) must be set up. The arguments are stored in argument buffers stored in L1B data memory. Specifically, the arguments for the Secure Function are stored at the top of L1B data memory, at address 0xFF90 0000. There are 24 bytes allocated for the arguments for the secure function. Following the argument buffer for the Secure Function is the argument buffer for the SESR, at address 0xFF90 0018. For security reasons this authentication protocol accesses fixed locations for arguments. When the user starts executing the Secure Function, it receives 2 arguments. The first argument (`R0`) contains the address of the Secure Function argument buffer. The second argument (`R1`) holds the `IMASK` value before shut off interrupts.

Secure Function Arguments

When the message is successfully authenticated, the program counter will vector to the Secure Function with the first argument (`R0`) containing a pointer to the top of the L1B data memory. The second argument (`R1`) of the secure function is the `IMASK` value. This value is obtained when the SESR successfully authenticates the message. Before the message is transferred via DMA to its final target run location, interrupts are shut off so tampering cannot occur between the time of successful authentication and execution of the secure function. The prototype for the secure function is:

```
void secure_function(tSecureFunctionArgs *, unsigned short
imask);
```

The 24-byte Secure Function argument buffer is for the convenience of user to be able to pass arguments to the Secure Function prior to starting authentication.

It is the responsibility of the user's Secure Function to re-enable interrupts by using the saved `IMASK` value or by using a new `IMASK` value.

The 24-byte Secure Function argument buffer can be used in any aligned fashion. For example, it can be used to store six 32-bit words or twelve 16-bit words, or any combination of data types such as integers, shorts and characters, as long as the accesses are aligned.

Secure Entry Service Routine Arguments

The argument buffer for the SESR is shown in Listing 16-3

Listing 16-3. Argument Buffer for SESR

```
/* SESR argument structure. Expected to reside at address
0xFF900018 */
typedef struct SESR_args {

unsigned short usFlags;/* security firmware flags*/
unsigned short usIRQMask;/* interrupt mask*/
unsigned long ulMessageSize;/* message length in bytes*/
unsigned long ulSFEntryPoint;/* entry point of secure function*/
unsigned long ulMessagePtr;/* pointer to the buffer containing
the digital signature and message */
unsigned long ulReserved1;/* reserved*/
unsigned long ulReserved2;/* reserved*/
} tSESR_args;
```

usFlags

The first argument, `usFlags`, is a 16bit bit flag that signals authentication what to do. Figure Figure 16-4 shows the meaning of the bits.

Bit 0 tells the authentication firmware whether or not to drop the interrupt level. To execute `raise 2;`, the Blackfin processor must operate in supervisor mode, in other words, operate at one of the interrupt levels. NMI must be asserted when authentication is initiated. The caller/user has the option to deassert NMI and drop back down to a lower interrupt level (the interrupt level in effect when NMI was asserted to initiate authentication) or continue authentication at NMI level.

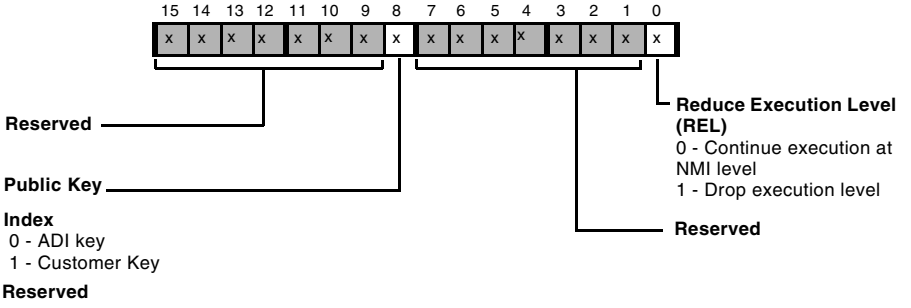


Figure 16-4. Bit Fields for Flags Argument

By lowering the interrupt level at which the authentication firmware executes, other interrupts can be serviced. Be aware that if another interrupt is serviced and the PC vectors out of the authentication firmware during authentication, the authentication process fails and returns an error code.

Bit 1 in the flags argument tells the authentication firmware whether or not to move the message/code to a final code space where it will be executed. This is only valid on certain ADSP-BF54x processor processors with L2 memory. Processors that have no L2 memory must move the message/code from L1A data memory to L1 code memory. The moves are done via memory DMA and are executed by the firmware.

Bit 8 tells the firmware which public key is used for authentication. The OTP memory holds two public keys. One is programmed by Analog Devices for failure analysis purposes only and the other is programmed by the developer.

usIRQMask

The `usIRQMask` argument is a 16-bit user-defined bitmask to be loaded into the lower 16 bits of the `IMASK` MMR if the execution level is to be lowered from NMI level. This argument allows the user to specify which,

if any, interrupts will be allowed to be serviced should they occur during the time authentication occurs. Note that if any interrupt is serviced, the authentication process fails and returns an error code as mentioned above. For more information regarding `IMASK`, refer to *Blackfin Processor Programming Reference*.

ulMessageSize

The `ulMessageSize` argument is a 32-bit non-negative integer that tells the SESR how big the message is, in bytes. The `ulMessageSize` must be a multiple of two, otherwise the SESR returns an error code.

ulSFEntryPoint

The `ulSFEntryPoint` argument is the final address that the message will be moved-to and executed-from. Again, since the authentication firmware expects code as the first portion of the message, the address must be a multiple of four since instructions can be either 16 bit or 32 bit lengths. If the message consists of both code and data, it is the user's responsibility to move the data to the proper area of data memory for subsequent use within their application.

ulMessagePtr

The `ulMessagePtr` argument holds the address where the digital signature and message are found.

Secure Message Execution

If the authentication of the digital signature is successful, the authentication firmware will directly vector the program counter to the Secure Function at its final target location, plus an offset of four bytes. The offset provides a location for the overlay ID if overlays are used with Lockbox. To return to the calling function, the authenticated message must execute

`rtn;` if execution level was not signaled to be lowered in the authentication firmware. Otherwise, if the execution level was lowered, the Secure Function can return via `rts;`.

To prevent tampering, interrupts and the watchdog timer are shut off near the end of successful authentication. It is the user's responsibility to re-enable the interrupts and the watchdog timer in the secure function if they are required in the user's application.

Return Codes

If for any reason, an error occurs, the SESR returns an error code and bit 7 in the `SECURE_STAT` MMR sets to indicate that register `R0` contains a valid error code. Table 16-2 lists a portion of the valid return codes.

Table 16-2. List of Return Codes from SESR

Return Codes	Value	Description
SECFW_SUCCESS	0	Success
SECFW_ERROR_INV_FLAGS	-1	"Flags" argument to firmware is invalid.
SECFW_ERROR_INV_INTMASK	-2	IRQ mask specified is invalid.
SECFW_ERROR_INV_CODESZ	-3	Code size specified is either non-positive or odd.
SECFW_ERROR_OOB_CODE	-6	The message (Secure function) is too big and surpasses the protected region in L1A.
SECFW_ERROR_BAD_EVT	-10	One of the ISR specified in the Event Vector table points inside the authentication firmware.
SECFW_ERROR_PUBKEY_ZERO	-11	Invalid public key of (0,0).
SECFW_ERROR_AUTH_FAILED	-12	Invalid message/signature pair.
SECFW_ERROR_DMA	-15	MDMA error occurred during DMA transfer or the message to the final target vector.

Table 16-2. List of Return Codes from SESR (Cont'd)

Return Codes	Value	Description
SECFW_ERROR_DROPPING_INT_FAILED	-17	Could not drop interrupt level from NMI.
SECFW_ERROR_FUSE_READ_FAILED	-18	Error occurred while reading OTP memory.
SECFW_ERROR_TGTVECT_NONALIGNED	-19	Target vector is not 4 Byte aligned.
SECFW_ERROR_SECURE0_WRITE_FAILED	-20	Write to Secure0 bit failed. Secure State Machine might be blocking the write because ISR was taken.
SECFW_ERROR_SM_NOT_ENTERED	-21	Secure0 bit was written three times but secure mode was still not entered.
SECFW_ERROR_BAD_TGT_ADDR	-22	Target vector must be in L1 code space or L2 (for ADSP-BF54x processor).
SECFW_ERROR_SF_TOO_BIG	-23	Message (Secure function) too big to fit at target location.

In addition to the return codes listed in Table 16-2, a return value between -62 to -252 is also a valid error return code. These errors are from OTP accesses.

To decipher the error from an OTP access, there is an offset that must be added to the error code. The macro `OTP_READ_ERROR_OFFSET` (defined in CCES or VisualDSP++ header files with a value of -285) is added to the return value. The result is a bit mask. Figure 16-5 shows the definition of the bit fields.

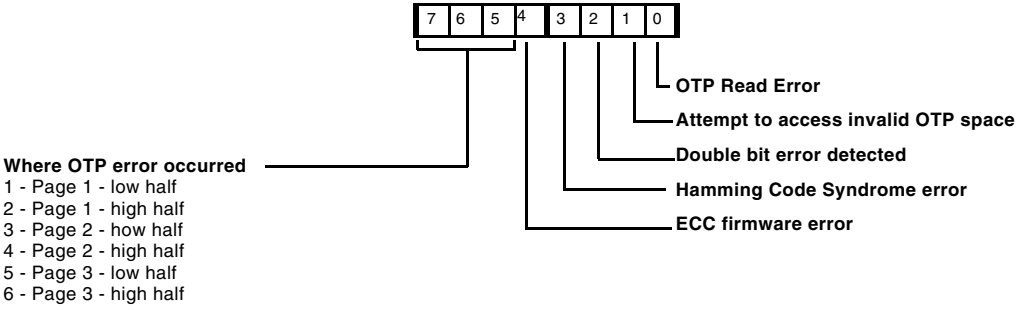


Figure 16-5. Bit Field Definition Return Value if OTP Error Occurred

Advanced Encryption Standard (AES) API

The ADSP-BF54x processor family of processors include a software implementation of the Advanced Encryption Standard (AES) as defined by the FIPS 197 publication in L1 ROM. This implementation of the AES symmetric-cipher is C-callable.

The following describes the application programming interface (API) for using AES including both data types and ROM routines.

ADI_AES_DATA Data Type

```
typedef struct ADI_AES_DATA
{
u32  *pKeyExpTmp;
u32  *pKR;
u32  *pState;
u32  *pIV;
u32  *pRcon;
u16  *pStateShiftExtract;
u16  *pInvStateShiftExtract;
u8   *pSBox;
u8   *pSBoxMixC;
u8   *pGFMpyTbl;
```

```
u8   *pInvSBox;
u8   *pInvSBoxMixC;
u32  *pStatePointers;
} ADI_AES_DATA;
```

The AES initialization routine, `bfrom_AesInit()`, when provided with a reference to an object of type `ADI_AES_DATA`, will initialize some of the buffers specified in the object. The caller of `bfrom_AesInit()` has to allocate storage both for the object of type `ADI_AES_DATA` and for the buffers specified in that object.

Some of the buffers specified in `ADI_AES_DATA` are necessary only for encryption or only for decryption. Therefore, if only one of encryption or decryption is used, fewer buffers from `ADI_AES_DATA` need to be allocated.

Table 16-3 shows the buffers specified by `ADI_AES_DATA`, their sizes, and whether or not they are used in encryption and in decryption.

Table 16-3. Buffers Specified in `ADI_AES_DATA`

Buffer	Encryption	Decryption	Size (in bytes)
pKeyExpTmp	X	X	32
pKR	X	X	16
pState	X	X	16
pIV	X	X	16
pRcon	X	X	64
pStateShiftExtract	X	X	32
pInvStateShiftExtract	X	X	32
pSBox	X	X	256
pSBoxMixC	X		1024
pGFMpyTbl		X	1024
pInvSBox		X	256

Table 16-3. Buffers Specified in ADI_AES_DATA (Cont'd)

pInvSBoxMixC		X	1024
pStatePointers	X	X	64

ADI_AES_KEYEXPANSION Data Type

```
typedef struct ADI_AES_KEYEXPANSION
{
u8              *pCipherKey;
u8              *pRoundKeys;
u32             udKeySize;
ADI_AES_DATA    *pAesData;
} ADI_AES_KEYEXPANSION;
```

The AES key expansion routines, `bfrom_AesKeyexp()` and `bfrom_AesInvKeyexp()`, when provided with a reference to an object of type `ADI_AES_KEYEXPANSION`, will perform an AES key expansion on the `udKeySize-long` key stored in `pCipherKey`, and will store the resulting AES rounds keys in `pRoundKeys`. See Table 16-4 for elements in an object of type.

Table 16-4. Elements in an Object of Type ADI_AES_KEYEXPANISION

pCipherKey	Pointer to the cipher key buffer, which is expected to hold the 128, 192, or 256-bit AES key.
pRoundKeys	Pointer to a buffer allocated by the caller of the key expansion routines. This buffer will hold the rounds keys generated by the key expansion routines. Buffer size must be (4* Nb* (Nr +1) + 1) bytes, where Nb is the number of columns (32-bit words) comprising the state and Nr is the number of rounds, as described in the AES specification. According to the specification, Nb is fixed at 4 and Nr can be 10, 12 or 14.
udKeySize	The AES key size used (in multiple of 32-bit words). May take on the values 4, 6, and 8 to specify keys of size 128, 192, and 256-bits respectively.
pAesData	Pointer to an object of type ADI_AES_DATA, which is initialized through a call to bfrom_AesInit()

ADI_AES_CIPHER Data Type

```
typedef struct ADI_AES_CIPHER {
u8     *pInputData;
u8     *pOutputData;
u8     *pRoundKeys;
u32    udDataLength;
u8     *pInitVector;
u32    udKeySize;
u32    udMode;
ADI_AES_DATA  *pAesData;
} ADI_AES_CIPHER;
```

The AES cipher routines, `bfrom_AesCipher()` and `bfrom_AesInvCipher()`, when provided with a reference to an object of type `ADI_AES_CIPHER`, will encrypt/decrypt the data in `pInputData` and will store the output in `pOutputData`. See Table 16-5 for elements in an object of type.

Table 16-5. Elements in an Object of Type `ADI_AES_CIPHER`

pInputData	Pointer to the input data buffer. In the case of encryption, this buffer should contain plaintext. In the case of decryption, this buffer should contain ciphertext.
pOutputData	Pointer to the output data buffer. After encryption, this buffer will contain ciphertext. After decryption, this buffer will contain plaintext.
pRoundKeys	Pointer to a buffer containing the AES round keys, which are generated by the key expansion routines bfrom_AesKeyexp() and bfrom_AesInvKeyexp(). Buffer size must be (4* Nb* (Nr +1) + 1) bytes, where Nb is the number of columns (32-bit words) comprising the state and Nr is the number of rounds, as described in the AES specification. According to the specification, Nb is fixed at 4 and Nr can be 10, 12 or 14.
udData-Length	The length of the input data in multiples of blocks of size 128-bits (16-bytes) each.
pInitVector	Certain block cipher modes of operation require an initialization vector. When an initialization vector is necessary, pInitVector points to the buffer containing the initialization vector.
udKeySize	The AES key size used (in multiples of 32-bit words). May take on the values 4, 6, and 8 to specify keys of size 128, 192, and 256-bits respectively.

Table 16-5. Elements in an Object of Type ADI_AES_CIPHER (Cont'd)

udMode	udMode specifies the block cipher mode of operation. The supported modes are: BLOCK_CIPHER_MODE_ECB for electronic codebook mode BLOCK_CIPHER_MODE_CBC for cipher block chaining mode BLOCK_CIPHER_MODE_OFB for output feedback mode BLOCK_CIPHER_MODE_CTR for counter mode
pAesData	Pointer to an object of type ADI_AES_DATA, which is initialized through a call to bfrom_AesInit()

bfrom_AesInit() ROM Routine

Entry address:
0xFFA1 4028

Arguments:
R0: Flags
R1: Pointer to an object of type ADI_AES_DATA

C prototype:

```
void bfrom_AesInit (u32 udFlags, ADI_AES_DATA *pAesData);
```

Return values:
AES_BOTH
AES_DECRYPTION
AES_ENCRYPTION

This function initializes the data buffers, which are referenced in ADI_AES_DATA and allocated by the caller, for use by the AES module.

This function is called first before other calls to the AES module.

Certain buffers are only necessary for encryption or decryption. Therefore, storage space may be saved by only allocating required buffers. The first argument specifies whether the user wishes to only encrypt, to only decrypt, or to both encrypt and decrypt. Table 16-3 lists the buffers referenced by ADI_AES_DATA, their sizes, and whether or not they are necessary for encryption and/or for decryption. If, for example, the user only needs

to encrypt data (no decryption necessary), then the user can specify AES_ENCRYPTION as the first argument to bfrom_AesInit(), thus eliminating the need to allocate the buffers pGFMpyTbl, pInvSBox, and pInvSBoxMixC.

bfrom_AesKeyexp() ROM Routine

Entry address:
0xFFA1 402C

Arguments:
R0: Pointer to an object of type ADI_AES_KEYEXPANSION

C prototype:
```
s32 bfrom_AesKeyexp (ADI_AES_KEYEXPANSION *pAesKeyexpData);
```

Return values:
AES_SUCCESS
AES_INVALID_KEY_SIZE

This function produces the round keys for the forward cipher from the cipher key. It should be called before executing bfrom_AesCipher().

bfrom_AesKeyexp() should be called every time a new cipher key is used. If, for example, several data buffers need to be encrypted and all of them need to be encrypted using the same key then only one call to bfrom_AesKeyexp() is necessary. However, if each data buffer needs to be encrypted using a different key, then bfrom_AesKeyexp() should be called prior to calling bfrom_AesCipher() for each buffer encryption.

Notice that for the modes of operation BLOCK_CIPHER_MODE_OFB and BLOCK_CIPHER_MODE_CTR, encryption and decryption are identical. Therefore, in these modes bfrom_AesKeyexp() should be used instead of bfrom_AesInvKeyexp() to produce the round keys for the inverse cipher.

bfrom_AesInvKeyexp() ROM Routine

Entry address:
0xFFA1 4030

Arguments:
R0: Pointer to an object of type ADI_AES_KEYEXPANSION

C prototype:
```
s32 bfrom_AesInvKeyexp (ADI_AES_KEYEXPANSION *pAesKeyexpData);
```

Return values:
AES_SUCCESS
AES_INVALID_KEY_SIZE

This function produces the rounds keys for the inverse cipher from the cipher key. It should be called before executing bfrom_AesInvCipher().

bfrom_AesInvKeyexp() should be called every time a new cipher key is used. If, for example, several data buffers need to be decrypted and all of them need to be decrypted using the same key then only one call to bfrom_AesInvKeyexp() is necessary. However, if each data buffer needs to be decrypted using a different key, then bfrom_AesInvKeyexp() should be called prior to calling bfrom_AesInvCipher() for each buffer decryption.

Notice that for the modes of operation BLOCK_CIPHER_MODE_OFB and BLOCK_CIPHER_MODE_CTR, encryption and decryption are identical. Therefore, in these modes bfrom_AesKeyexp() should be used instead of bfrom_AesInvKeyexp() to produce the round keys for the inverse cipher.

bfrom_AesCipher() ROM Routine

Entry address:
0xFFA1 4020

Arguments:
R0: Pointer to an object of type ADI_AES_CIPHER

C prototype:
```
s32 bfrom_AesCipher (ADI_AES_CIPHER *pAesCipherData);
```

Return values:
```
AES_SUCCESS
AES_INVALID_MODE
```

This function performs the AES forward cipher operation.

Notice that for the modes of operation `BLOCK_CIPHER_MODE_OFB` and `BLOCK_CIPHER_MODE_CTR`, encryption and decryption are identical. Therefore, in these modes `bfrom_AesCipher()` should be used instead of `bfrom_AesInvCipher()` to perform the AES inverse cipher operation.

bfrom_AesInvCipher() ROM Routine

Entry address:
0xFFA1 4024

Arguments:
`R0`: Pointer to an object of type `ADI_AES_CIPHER`

C prototype:
```
s32 bfrom_AesInvCipher (ADI_AES_CIPHER *pAesCipherData);
```

Return values:
```
AES_SUCCESS
AES_INVALID_MODE
```

This function performs the AES inverse cipher operation.

Notice that for the modes of operation `BLOCK_CIPHER_MODE_OFB` and `BLOCK_CIPHER_MODE_CTR`, encryption and decryption are identical. Therefore, in these modes `bfrom_AesCipher()` should be used instead of `bfrom_AesInvCipher()` to perform the AES inverse cipher operation.

SECURE HASH ALGORITHM (SHA-1) API

The ADSP-BF54x processor processor includes a software implementation of the Secure Hash Algorithm (SHA-1) in L1 ROM. This implementation of the SHA-1 hash algorithm is C-callable.

The following describes the application programming interface (API) for using SHA-1 including both data types and ROM routines.

ADI_SHA1 Data Type

```
typedef struct ADI_SHA1 {
u8   *pInputMessage;
u32  udMessageSize;
u8   *pOutputDigest;
u8   *pScratchBuffer;
} ADI_SHA1;
```

The SHA1 hash routine, `bfrom_Sha1Hash`, when provided with a reference to an object of type `ADI_SHA1`, hashes the `udMessageSize-long` message referenced by `pInputMessage`, and stores the hash value (also referred to as message digest) in the buffer referenced by `pOutputDigest`. The elements in an object of type `ADI_SHA1`, are shown in Table 16-6.

Table 16-6. Elements in an Object of Type `ADI_SHA1`

pInputMessage	Pointer to the input buffer.
udMessageSize	The size, in bytes, of the valid input data in pInputMessage.
pOutputDigest	Pinter to the output data buffer. After hashing, this buffer will contain the digest of the input message. The digest is 160-bits (SHA1_HASH_SIZE-bytes) long
pScratchBuffer	Pointer to a data buffer of size, SHA1_SCRATCH_BUFFER_SIZE-bytes, used by the SHA-1 module.

bfrom_Sha1Init ROM Routine

Entry address:
0xFFA1 4024

Arguments:
`R0`: Pointer to a buffer of size `SHA1_SCRATCH_BUFFER_SIZE`

C prototype:
```
void bfrom_Sha1Init (u8 *pScratchBuffer);
```

This function initializes some data elements in `pScratchBuffer`. It is called first before making any calls to `bfrom_Sha1Hash`.

bfrom_Sha1Hash ROM Routine

Entry address:
0xFFA1 4024

Arguments:
`R0`: Pointer to an object of type `ADI_SHA1`

C prototype:
```
void bfrom_Sha1Hash (ADI_SHA1 *pSha1);
```

This function performs the hash operation.

ARC4 API

The ADSP-BF54x processor processor includes a software implementation of the ARC4 algorithm in L1 ROM. This implementation of ARC4 is C-callable.

The following describes the application programming interface (API) for using ARC4 including both data types and ROM routines.

ADI_ARC4_KEY Data Type

```
typedef struct  ADI_ARC4_KEY {
  u32  *pSBox;
  u32  *pKey;
  u32  udKeyLength;
} ADI_ARC4_KEY;
```

See Table 16-7 for elements in an object of type `ADI_ARC4_KEY`.

Table 16-7. Elements in an Object of Type `ADI_ARC4_KEY`

pSBox	Pointer to an ARC4 substitution box.
pKey	A pointer to a buffer containing the ARC4 key.
udKeyLength	The size of the ARC4 key specified in pKey.

ADI_ARC4_DATA Data Type

```
typedef struct ADI_ARC4_DATA {
  u32    *pSBox;
  u32    *pData;
  u32    udDataLength;
} ADI_ARC4_DATA;
```

See Table 16-8 for elements in an object of type `ADI_ARC4_DATA`.

Table 16-8. Elements in an Object of Type `ADI_ARC4_DATA`

pSBox	Pointer to an ARC4 substitution box, which has already been initialized through a call to bfrom_Arc4Init().
pData	A pointer to a buffer containing the data to be encrypted or decrypted. Notice that the ARC4 module performs encryption and decryption in-place. Therefore, after calling the ARC4 cipher function, bfrom_Arc4Cipher(), this buffer will contain the encrypted or decrypted output.
udDataLength	The size of the data pointed to by pData.

bfrom_Arc4Init ROM Routine

Entry address:
0xFFA1 4018

Arguments:
`R0`: Pointer to an object of type `ADI_ARC4_KEY`

C prototype:
```
void bfrom_Arc4Init (ADI_ARC4_KEY *pArc4Key)
```

The ARC4 initialization routine, `bfrom_Arc4Init()` initializes the buffer pointed to by `pSBox` based on the key specified in `pKey` and `udKeyLength`.

`bfrom_Arc4Init()` should be called first before executing `bfrom_Arc4Cipher()`.

bfrom_Arc4Cipher ROM Routine

Entry address:
0xFFA1 401C

Arguments:
`R0`: Pointer to an object of type `ADI_ARC4_DATA`

C prototype:
```
void bfrom_Arc4Cipher (ADI_ARC4_DATA *pArc4Data)
```

The ARC4 encryption/decryption routine, `bfrom_Arc4Cipher()`, encrypts/decrypts the data specified in `pData` and `udDataLength` using the substitution box specified in `pSBox`.

`bfrom_Arc4Cipher()` should be called after the substitution box has been initialized through a call to `bfrom_Arc4Init()`.

Security Registers

There are three registers which provide information that can be used during security mode control and to return status of the Secure State Machine states. These registers require privileged access depending on the operating state of the processor.

Table 16-9. Security Registers

Memory Mapped Address	Register	Size (Bits)	Description
0xFFC0 4320	SECURE_SYSSWT	32	"Secured System Switches (SECURE_SYSSWT) Register" on page 16-57
0xFFC0 4324	SECURE_CONTROL	16	"Secure Control (SECURE_CONTROL) Register" on page 16-64
0xFFC0 4328	SECURE_STATUS	16	"Secure Status (SECURE_STATUS) Register" on page 16-67

Secured System Switches (SECURE_SYSSWT) Register

Secured system switches control hardware that would otherwise allow a threat of attack to a secured system. Hardware is controlled voluntarily and involuntarily as follows:.

- During Open Mode the switches are involuntarily set with all controls off (unrestricted access) with exception of access to OTP protected “secrets” area. OTP secrets are always protected and can only be accessible upon entry into Secure Mode.
- During Secure Entry Mode all switches are initially set with all controls on (restricted access). Two exceptions are the OTP secrets control (OTPSEN bit) is not accessible so access to the secrets OTP area remains restricted and the RSTDABL bit remains deactivated (External Reset is allowed).
- During Secured Mode operation all switches are voluntary (initially set) and under the control of authenticated code. Therefore restricted access controls can be reconfigured by authenticated user code. This includes the activation of Reset Disable (RSTDABL bit).

The register, shown in Figure 16-6 on page 16-58 and Figure 16-7 on page 16-59, is 32-bits wide and requires 32-bit access. Limited write access to a few bits is allowed in Secure Entry mode, and full write access to all bits is allowed in Secure mode. No write access is allowed in Open Mode.

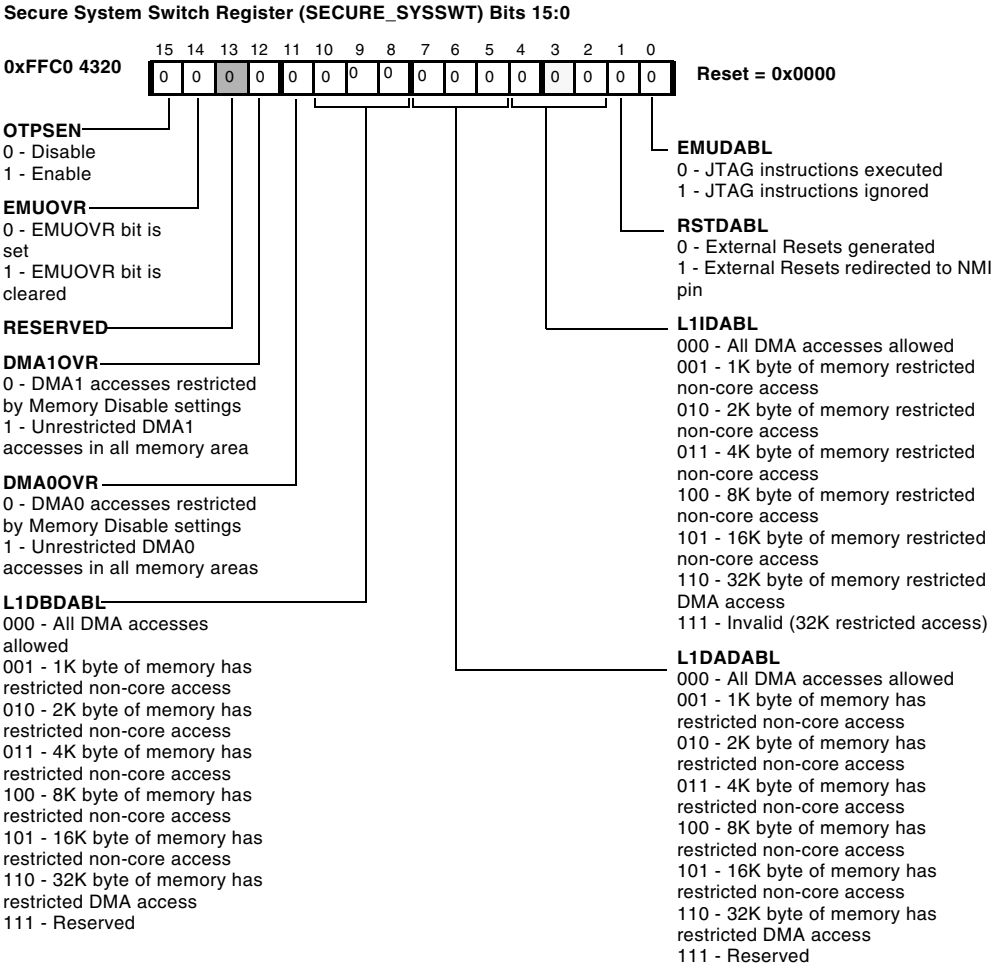


Figure 16-6. Secure System Switch Register, Bits 15:0

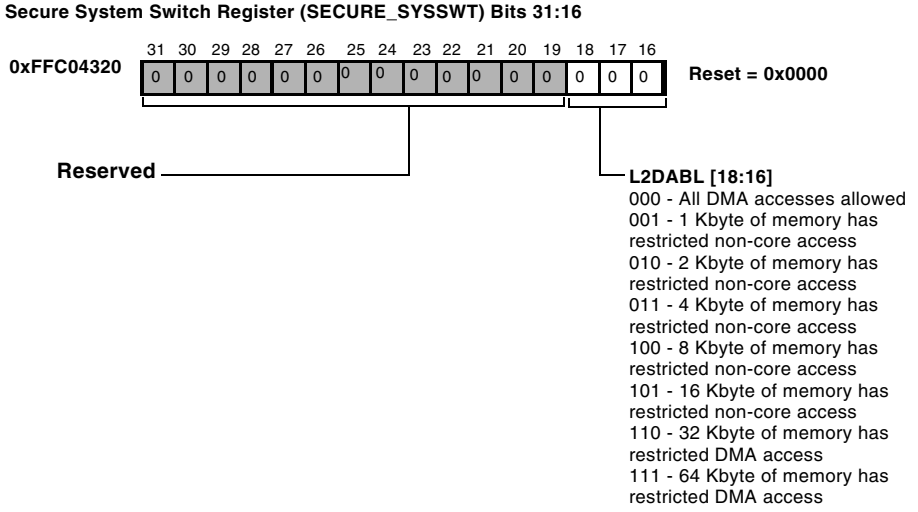


Figure 16-7. Secure System Switch Register, Bits 31:16

Table 16-10. Secure System Switch Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000 Secured Entry = 0x0007 04D9 Secured Mode = 0x0007 04DB
0	EMUDABL	Emulation Disable. Upon Secured Entry the EMUDABL setting is based on the previous state of EMUOVR. Upon re-entering Open Mode, EMUDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secured Mode. 0 - Analog Devices JTAG emulation instructions are recognized and executed. Once this bit is cleared while in Secured Mode it will not be set upon Secured Entry. This condition will remain until reset, at which time it is cleared. This feature is used in security debug. 1 - Analog Devices JTAG emulation instructions are ignored. Standard emulation commands such as bypass are allowed.

Table 16-10. Secure System Switch Register (Cont'd)

Bit Position	Bit Name	Bit Description
1	RSTDABL	Reset Disable. This bit is not effected upon Secured Entry. This bit is set upon entering Secured Mode. Upon re-entering Open Mode, RSTDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secured Mode. 0 - External Resets are generated and serviced normally. 1 - External Resets are redirected to the NMI pin. This avoids circumventing memory clean operations.
4:2	L1IDABL	L1 Instruction Memory Disable. Upon Secured Entry L1IDABL is set to 0x6. Upon re-entering Unsecured Mode, L1IDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 Instruction areas. 001 - 1K byte of memory (0xFFA0 0000 - 0xFFA0 03FF) has restricted non core access 010 - 2K byte of memory (0xFFA0 0000 - 0xFFA0 07FF) has restricted non core access 011 - 4K byte of memory (0xFFA0 0000 - 0xFFA0 0FFF) has restricted non core access 100 - 8K byte of memory (0xFFA0 0000 - 0xFFA0 1FFF) has restricted non core access 101 - 16K byte of memory (0xFFA0 0000 - 0xFFA0 3FFF) has restricted non core access 110 - 32K byte of memory (0xFFA0 0000 - 0xFFA0 7FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry. 111 - Reserved

Table 16-10. Secure System Switch Register (Cont'd)

Bit Position	Bit Name	Bit Description
7:5	L1DADABL	L1 Data Bank A Memory Disable. Upon Secured Entry L1DADABL is set to 0x6. Upon re-entering Open Mode, L1DADABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 data bank A areas. 001 - 1K byte of memory (0xFF80 0000 - 0xFF80 03FF) has restricted non core access 010 - 2K byte of memory (0xFF80 0000 - 0xFF80 07FF) has restricted non core access 011 - 4K byte of memory (0xFF80 0000 - 0xFF80 0FFF) has restricted non core access 100 - 8K byte of memory (0xFF80 0000 - 0xFF80 1FFF) has restricted non core access 101 - 16K byte of memory (0xFF80 0000 - 0xFF80 3FFF) has restricted non core access 110 - 32K byte of memory (0xFF80 0000 - 0xFF80 7FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry. 111 - Reserved

Table 16-10. Secure System Switch Register (Cont'd)

Bit Position	Bit Name	Bit Description
10:8	L1DBDABL	L1 Data Bank B Memory Disable. Upon Secured Entry L1DBDABL is set to 0x4 giving L1 Data Bank B 8 Kbyte of non core restricted access. Upon re-entering Open Mode, L1DBDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 data bank B areas. 001 - 1K byte of memory (0xFF90 0000 - 0xFF90 03FF) has restricted non core access 010 - 2K byte of memory (0xFF90 0000 - 0xFF90 07FF) has restricted non core access 011 - 4K byte of memory (0xFF90 0000 - 0xFF90 0FFF) has restricted non core access 100 - 8K byte of memory (0xFF90 0000 - 0xFF90 1FFF) has restricted non core access. This is the initial setting upon entering Secured Entry. 101 - 16K byte of memory (0xFF90 0000 - 0xFF90 3FFF) has restricted non core access 110 - 32K byte of memory (0xFF90 0000 - 0xFF90 7FFF) has restricted DMA access. 111 - Reserved
11	DMA0OVR	DMA0 Memory Access Override Entering Secured Entry or Secured Mode does not effect this bit. Upon re-entering Open Mode, DMA0OVR is cleared. This bit is always read accessible. This bit is write accessible in both Secured Entry and Secured Mode. Controls DMA0 access to L1 Instruction, L1 Data and memory other than L1 regions. When clear access restrictions are based on Memory Disable settings within this register. 0 - DMA0 accesses are restricted based on Memory Disable settings. 1 - Unrestricted DMA0 accesses are allowed to all memory areas.

Table 16-10. Secure System Switch Register (Cont'd)

Bit Position	Bit Name	Bit Description
12	DMA1OVR	DMA1 Memory Access Override Entering Secured Entry or Secured Mode does not effect this bit. Upon re-entering Open Mode, DMA1OVR is cleared. This bit is always read accessible. This bit is write accessible in both Secured Entry and Secured Mode. Controls DMA1 access to L1 Instruction, L1 Data and memory other than L1 regions. When clear access restrictions are based on Memory Disable settings within this register. 0 - DMA1 accesses are restricted based on Memory Disable settings. 1 - Unrestricted DMA1 accesses are allowed to all memory areas.
13	RESERVED	Reserved bit. This reserved bit always returns a "0" value on a read access. Writing this bit with any value has no effect.
14	EMUOVR	Emulation Override This bit is always read accessible. This bit may be written with a 1 in secured mode only. This bit can be cleared in any mode (Unsecured mode, Secured Entry and Secured mode). Controls the value of EMUDABL upon Secured Entry. 0 - Upon Secured Entry the EMUDABL bit is set. 1 - Upon Secured Entry the EMUBABL bit is cleared. This bit can only be set when EMUDABL (bit-0) is written with a "0" while this bit (bit-14) is simultaneously written with a 1.

Table 16-10. Secure System Switch Register (Cont'd)

Bit Position	Bit Name	Bit Description
15	OTPSEN	OTP Secrets Enable. This bit can be read in all modes but is write accessible in Secured Mode only. 0 - Read and Programming access of the "secured" OTP Fuse area is restricted. Accesses will result in an access error (FERROR) 1 - Read and Programming access of the "secured" OTP Fuse area is allowed. If the corresponding program protection bit for an access is set, a program access is protected regardless of this bit's setting.
18:16	L2DABL	L2 Disable. Upon Secured Entry L2DABL is set to 0x7. Upon re-entering Open Mode, L2DABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L2. 001 - 1K byte of memory (0xFEB0 0000 - 0xFEB0 03FF) has restricted non core access 010 - 2K byte of memory (0xFEB0 0000 - 0xFEB0 07FF) has restricted non core access 011 - 4K byte of memory (0xFEB0 0000 - 0xFEB0 0FFF) has restricted non core access 100 - 8K byte of memory (0xFEB0 0000 - 0xFEB0 1FFF) has restricted non core access 101 - 16K byte of memory (0xFEB0 0000 - 0xFEB0 3FFF) has restricted non core access 110 - 32K byte of memory (0xFEB0 0000 - 0xFEB0 7FFF) has restricted non core access 111 - 64K byte of memory (0xFEB0 0000 - 0xFEB0 FFFF) has restricted DMA access. This is the initial setting upon entering Secured Entry.

Secure Control (SECURE_CONTROL) Register

The SECURE_CONTROL register is used during Secure Entry Mode authentication. This register is used to establish Secure Mode transition and can be used at any time to exit from Secure Mode. The register, shown in Figure 16-8, is 16-bits wide and requires 16-bit access.

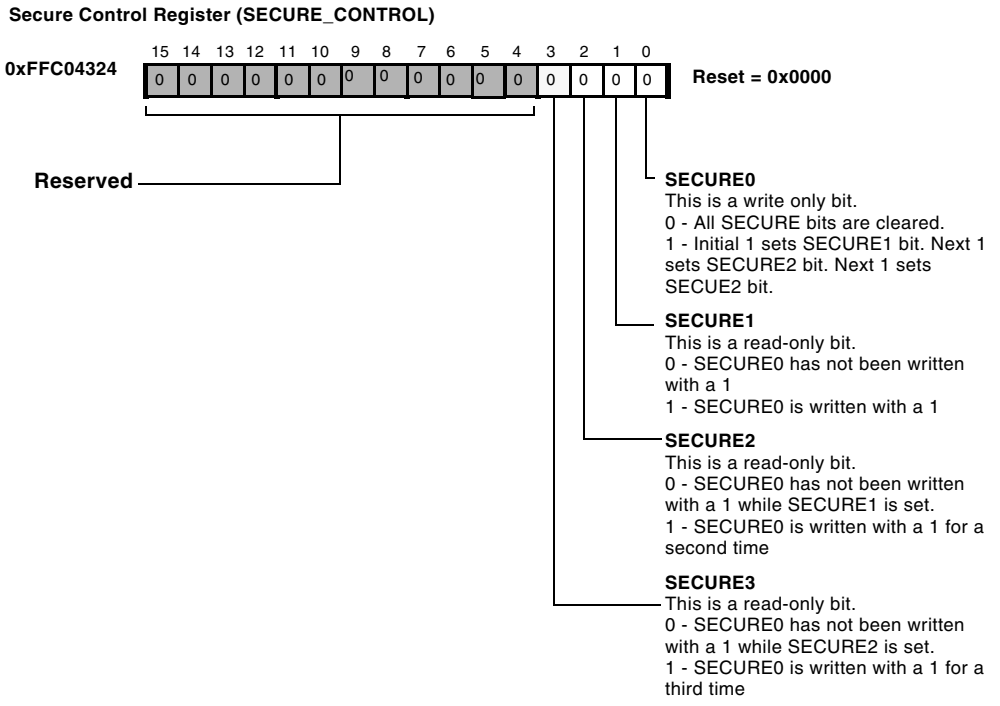


Figure 16-8. Secure Control Register

Table 16-11. Secure Control Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
0	SECURE0	SECURE 0 This is a write only bit. A read always returns "0". A 1 value can only be written to SECURE0 when in Secured Entry. The purpose of this control bit is to require 3 successive writes with a value of 1 to SECURE0 in order to enter Secured Mode. 0 - When written with a "0" value, all SECURE bits within this register are cleared and Open Mode is entered. All SYSSWT bits are cleared with the exception of EMUOVR. If EMUOVR had been set by the user, it will remain set (until RESET is asserted or until it is written with a "0"). 1 - Initially when written with a 1 value SECURE1 is set. With a subsequent 1 written SECURE2 is set. A subsequent 1 written will set SECURE3. Upon a set of SECURE3 Secured Mode is entered.
1	SECURE1	SECURE 1 This is a read-only bit and indicates a successful write of SECURE0 with a data value of 1 0 - SECURE0 has not been written with a 1 value 1 - SECURE0 is written with a 1 value
2	SECURE2	SECURE 2 This is a read-only bit and indicates two successful writes of SECURE0 with a data value of 1 has occurred 0 - SECURE0 has not been written with a 1 value while SECURE1 was set. 1 - SECURE0 is written with a 1 value for a second time.
3	SECURE3	SECURE 3 This is a read-only bit and indicates three successful writes of SECURE0 with a data value of 1 has occurred. 0 - SECURE0 has not been written with a 1 value while SECURE2 was set 1 - SECURE0 is written with a 1 value for a third time. The part is currently in Secured Mode and the SYSSWT register is writable by Authenticated code.

SECURE0 bit is user accessible and is used to exit from Secure Mode. Bits SECURE1, SECURE2, and SECURE3 are not user accessible and are accessed only by the firmware during the digital signature validation process.

Secure Status (SECURE_STATUS) Register

The SECURE_STATUS register provides information about the current secure state. This information can be used during security mode control as well as understanding why an authentication attempt has failed. The register, shown in Figure 16-9, is 16-bits wide and requires 16-bit access.

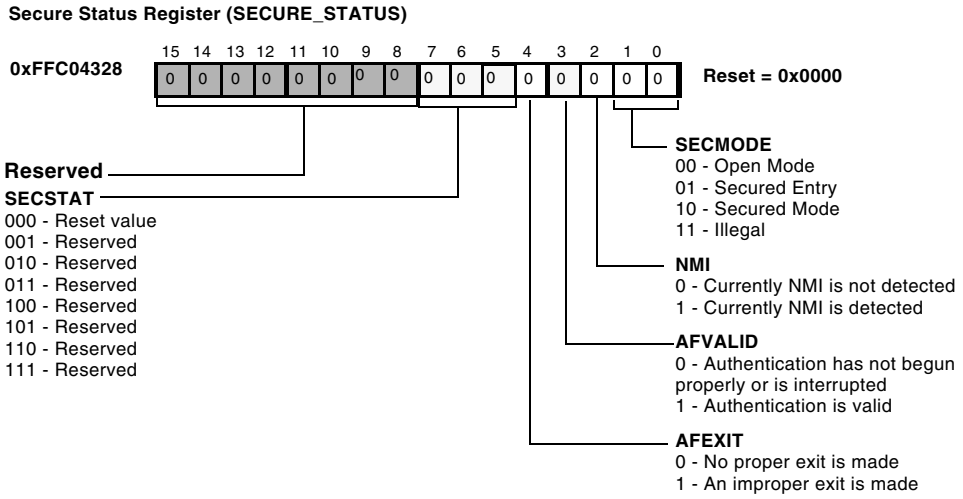


Figure 16-9. Secure Status Register

Table 16-12. Secure Status Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
1:0	SECMODE	Secured Mode Control State This are read-only bits that reflects the current Secure Mode Control's state. 00 - Open Mode 01 - Secured Entry 10 - Secured Mode 11 - Illegal
2	NMI	This is a read-only bit that reflects the detection of NMI. 0 - Currently NMI is not detected. 1 - Currently NMI is detected.
3	AFVALID	Authentication Firmware Valid This is a read-only bit that reflects the state of the Real Time Trace logic. If execution of authentication has begun properly and has had un interrupted operation the authentication is considered valid. A valid authentication is required for Secured Entry and Secured Mode operation. 0 - Authentication has not begun properly or is interrupted. 1 - Authentication is valid and is progressing properly and uninterrupted.
4	AFEXIT	Authentication Firmware Exit This is a write one to clear status bit. In the event authentication has begun properly but has had an improper exit before completion, this bit is set. This can only occur on an exit from Secured Entry back to Open Mode. 0 - No improper exit is made while executing authentication firmware. 1 - An improper exit from authentication firmware is made.
7:5	SECSTAT	Secure Status These are some read write bits which is defined later. These are intended to pass a status back to the handler in the event an authentication has failed. 000 - Reset value 001 - Reserved 010 - Reserved 011 - Reserved 100 - Reserved 101 - Reserved 110 - Reserved 111 - Reserved

Authentication Firmware Valid (AFVALID) is an input to the Secure State Machine and not an output control/status. AFVALID goes active based on reaching the correct program counter address.

17 SYSTEM RESET AND BOOTING

This document contains material that is subject to change without notice. The content of the boot ROM as well as hardware behavior may change across silicon revisions. See the anomaly list for differences between silicon revisions.

Overview

When the $\overline{\text{RESET}}$ input signal releases, the processor starts fetching and executing instructions from the on-chip boot ROM at address 0xEF00 0000.

The internal boot ROM includes a small boot kernel that loads application data from an external memory or host device. The application data is expected to be available in a well-defined format called the boot stream. A boot stream consists of multiple blocks of data and special commands that instruct the boot kernel how to initialize on-chip L1 and L2 SRAM memories as well as off-chip volatile memories.

The boot kernel processes the boot stream block-by-block until it is instructed by a special command to terminate the procedure and jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

The processor features four dedicated input pins BMODE[3:0] that select the booting mode. The boot kernel evaluates the BMODE pins and performs booting from respective sources. Table 17-1 describes the modes of the BMODE pins.

Table 17-1. Booting Modes

BMODE[3:0]	Boot Source	Description
0000	No boot – idle	The processor does not boot. Rather, the boot kernel executes an IDLE instruction.
0001	Boot from 8-bit or 16-bit external flash memory	The kernel boots from address 0x2000 0000 in asynchronous memory bank 0. The first byte of the boot stream contains further instructions whether the memory is eight or 16 bits wide.
0010	Boot from 16-bit asynchronous FIFO	By using the handshaked memory DMA (HMDMA1) feature through the $\overline{\text{DMAR1}}$ input, the kernel boots from address 0x2030 0000 in asynchronous memory bank 3.
0011	Boot from serial SPI memory	After an initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit or 32-bit addressable SPI flash or EEPROM memory that connects to $\overline{\text{SPI0\_SEL1}}$.
0100	Boot from SPI host	In this slave mode, the kernel expects the boot stream to be applied to SPI0 by an external host device.
0101	Boot from serial TWI memory	The kernel boots from TWI memory connected to TWI0. Memory is expected to respond to the unique slave identifier of 0xA0.
0110	Boot from TWI host	In this slave mode, the kernel expects the boot stream to be applied to TWI0 by an external host device. The Blackfin processor uses the slave identifier 0x5F.

Table 17-1. Booting Modes (Cont'd)

BMODE[3:0]	Boot Source	Description
0111	Boot from UART1 host	In this slave mode, the kernel expects the boot stream to be applied to UART1 by an external host device. The UART1RTS output is active and controlled by hardware. Prior to providing the boot stream, the host device is expected to send a 0x40 (ASCII '@') character that is examined by the kernel to adjust the bit rate.
1000	Reserved	
1001	Reserved	
1010	Boot from SDRAM memory[1]	This mode provides a quick warm boot option. It requires the SDRAM controller to be programmed by the preboot routine based on OTP settings. The kernel starts booting from address 0x0000 0010.
1011	Boot from on-chip OTP memory	This is the only stand-alone booting mode. It boots from the on-chip serial OTP memory. By default, the boot stream is expected to reside from OTP page 0x40 on. The start page can be altered by programming the OTP_START_PAGE field in OTP page PBS01H.
1100	Reserved	
1101	Boot from 8- and 16-bit NAND flash.	The boot kernel automatically detects whether an 8-bit small-page device or an 8-/16-bit large-page device is connected to the NFC. The NAND flash may optionally contain further initialization code that enables some more advanced boot options.

Table 17-1. Booting Modes (Cont'd)

BMODE[3:0]	Boot Source	Description
1110	Boot from 16-bit Host DMA	The kernel initializes the Host DMA unit to 16-bit ACK mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.
1111	Boot from 8-bit Host DMA	The kernel initializes the Host DMA unit 8-bit INT mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.

1 This chapter uses the term SDRAM as a synonym for off-chip synchronous dynamic memory. For the ADSP-BF54x products, SDRAM memory complies with either the DDR1 SDRAM or the Mobile DDR1 SDRAM standard.

Reset and Power-up

There is a subroutine in the boot kernel known as *preboot*, which is executed prior to the boot mode being processed. This preboot routine can customize default values of MMR registers, such as the PLL and SDRAM controller registers. Furthermore, the SPI and TWI master modes can be customized. The preboot behavior is controlled through OTP programming.

To enable booting from volatile memories such as SDRAM, the SDRAM controller must be programmed *before* data can be loaded into the memory. Either the preboot or the initialization code mechanism can be used for this purpose.

Table 17-2 describes the six types of resets.

Each type resets the core except for the System Software reset.

Table 17-2. 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 bits [15:4] of the SYSCR register. For more information, see "System Reset Configuration (SYSCR) Register" on page 17-105.
Wake up from hibernate state	Wake-up event as enabled in the VR_CTL register and reported by the PLL_STAT register.	Behaves as hardware reset except the WURESET bit in the SYSCR register is set. Booting can be performed conditionally on this event.
System software reset	Calling the bfrom_SysControl() routine with the SYSCTRL_SYSRESET option triggers a system reset.	Resets only the peripherals, excluding the RTC (real time clock) block and most of the DPMC. The system software reset clears bits [15:13] and bits [11:4] of the SYSCR register, but not the WURESET bit. The core is not reset and a boot sequence is not triggered. Sequencing continues at the instruction after bfrom_SysControl() returns.
Watchdog timer reset	Programming the watchdog timer causes a watchdog timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. (Because of the partial reset to the DPMC, the watchdog timer reset is not functional when the processor is in Sleep or Deep Sleep modes.) The SWRST or the SYSCR register can be read to determine whether the reset source was the watchdog timer.

Table 17-2. Resets (Cont'd)

Reset	Source	Result
Core double-fault reset	A core double fault occurs when an exception happens while the exception handler is executing. If the core enters a double-fault state, a reset can be caused by unmasking the DOUBLE_FAULT bit in the SWRST register.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST or SYSCR registers can be read to determine whether the reset source was a core double-fault.
Software reset	This reset is caused by executing a RAISE 1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) through emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Program executions vector to the 0xEF00 0000 address. The boot code executes an immediate system reset to ensure system consistency.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted after a specified asserted hold time to perform a hardware reset. For more information, see *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

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

The BMODE pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DDEXT} or GND. The pins and the corresponding bits in the SYSCR register configure the boot mode that is employed after hardware reset or system software reset. See *Blackfin Processor Programming Reference* for further information.

Software Resets

A software reset may be initiated in three ways.

- By the watchdog timer, if appropriately configured
- Calling the `bfrom_SysControl()` API function residing in the on-chip ROM. For further information, see Chapter 18, "Dynamic Power Management".
- By the `RAISE 1` instruction

The watchdog timer resets both the core and the peripherals, as long as the processor is in Active or Full-On mode. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

In order to perform a system reset, the `bfrom_SysControl()` routine must be called while executing from L1 memory (either as cache or as SRAM). When L1 instruction memory is configured as cache, make sure the system reset sequence is read into the cache.

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

For a reset generated by formatting the watchdog timer, the processor transitions into the boot mode sequence. The boot mode is configured by the state of the `BMODE` bit field in the `SYSCR` register.

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

A software reset only affects the state of the core. The boot kernel immediately issues a system reset to keep consistency with the system domain.

Reset Vector

When reset releases, the processor starts fetching and executing instructions from address 0xEF00 0000. This is the address where the on-chip boot ROM resides.

On a hardware reset, the boot kernel initializes the EVT1 register to 0xFFA0 0000. When the booting process completes, the boot kernel jumps to the location provided by the EVT1 vector register. With the exception of the HOSTDP boot modes, the content of the EVT1 register is overwritten by the TARGET ADDRESS field of the first block of the applied boot stream. If the BCODE field of the SYSCR register is set to 1 (no boot option), the EVT1 register is not modified by the boot kernel on software resets. Therefore, programs can control the reset vector for software resets through the EVT1 register. This process is illustrated by the flow chart in Figure 17-1.

The content of the EVT1 register may be undefined in emulator sessions.

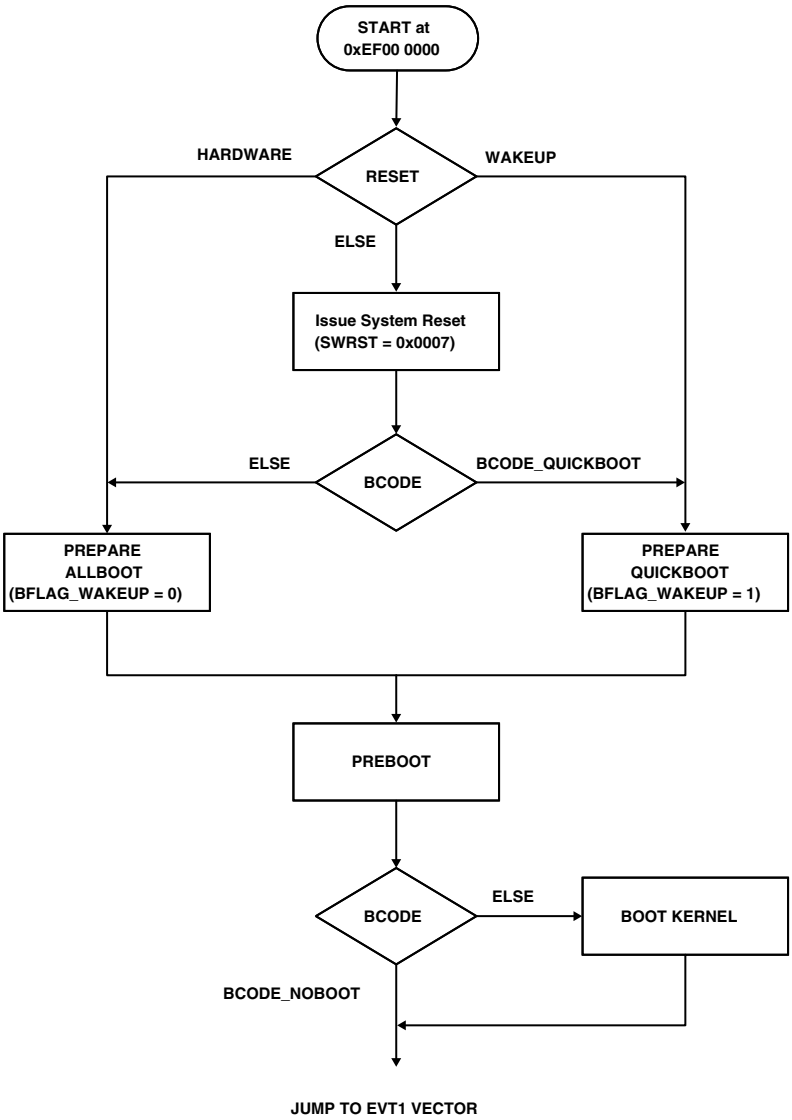


Figure 17-1. Global Boot Flow

Servicing Reset Interrupts

The processor services a reset event like other interrupts. The reset interrupt has top priority. Only emulation events have higher priority. When coming out of reset, the processor is in supervisor mode and has full access to all system resources. The boot kernel can be seen as part of the reset service routine. It runs at the top interrupt priority level.

Even when the boot process has finished and the boot kernel passes control to the user application, the processor is still in the reset interrupt. To enter user mode, the reset service routine must initialize the `RETI` register and terminate with an `RTI` instruction.

For a programming example, see "System Reset" on page 17-145.

Listing 17-1 and Listing 17-2 on page 17-146 show code examples that handle the reset event. See *Blackfin Processor Programming Reference* for details on user and supervisor modes.

Systems that do not work in an operating system environment may not enter user mode. Typically, the interrupt level needs to be degraded down to IVG15. Listing 17-3 and Listing 17-4 on page 17-147 show how this is accomplished.

Since the boot kernel is running at reset interrupt priority, NMI events, hardware errors and exceptions are not serviced at boot time. As soon as the reset service routine returns, the processor can service the events that occurred during the boot sequence. It is recommended that programs install NMI, hardware error, and exception handlers before leaving the reset service routine. This includes proper initialization of the respective event vector registers `EVTx`.

Preboot

After reset, the boot kernel residing in the on-chip boot ROM does not immediately start processing the boot stream. First it calls a subroutine called preboot, as shown in Figure 17-2 on page 17-12 and Figure 17-3 on page 17-13. The preboot routine customizes the default values of several system MMR registers based on user-configurable OTP (one-time programmable) memory. The following modules can be customized in this way.

- PLL and voltage regulator settings
- SDRAM controller settings
- Asynchronous EBIU settings

Some OTP bits customize the boot process:

- Bit rate of SPI and TWI master boot modes
- TWI master boot addressing scheme
- Activation of SPI fast read mode
- Boot host wait (`HWAIT`) signal

Further OTP bits let the user disable certain features of the processor:

- Individual boot modes (for security reasons)

Finally, certain bits are already preset in the factory:

- USB voltage trim
- Individual boot modes

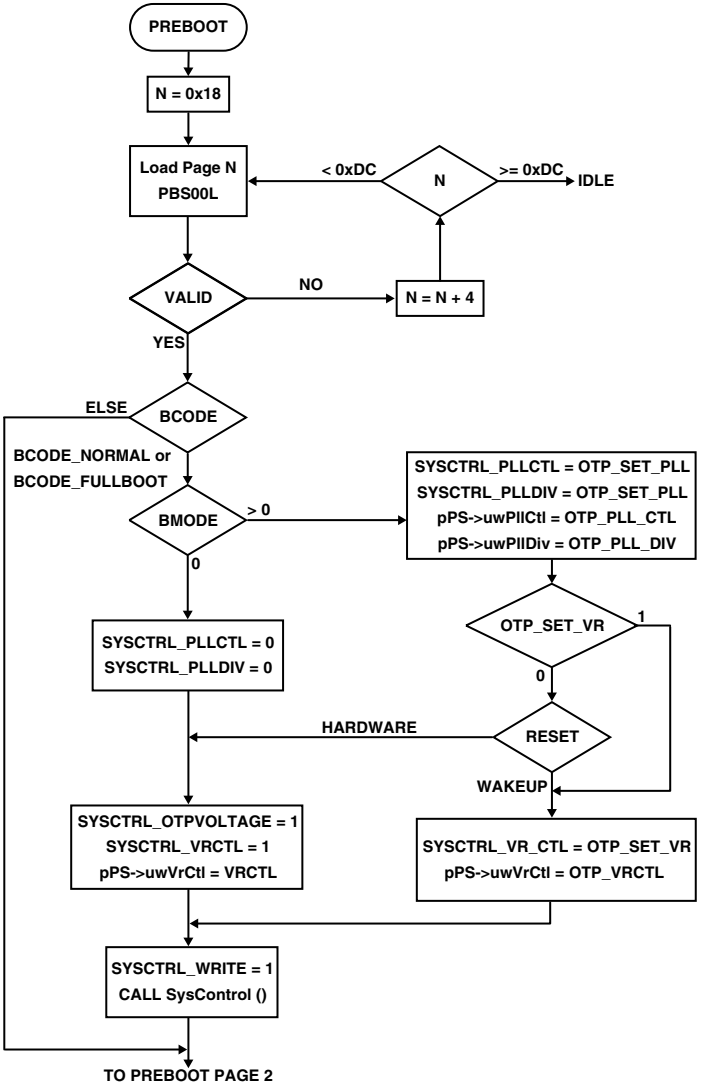


Figure 17-2. Preboot Flow 1 of 2

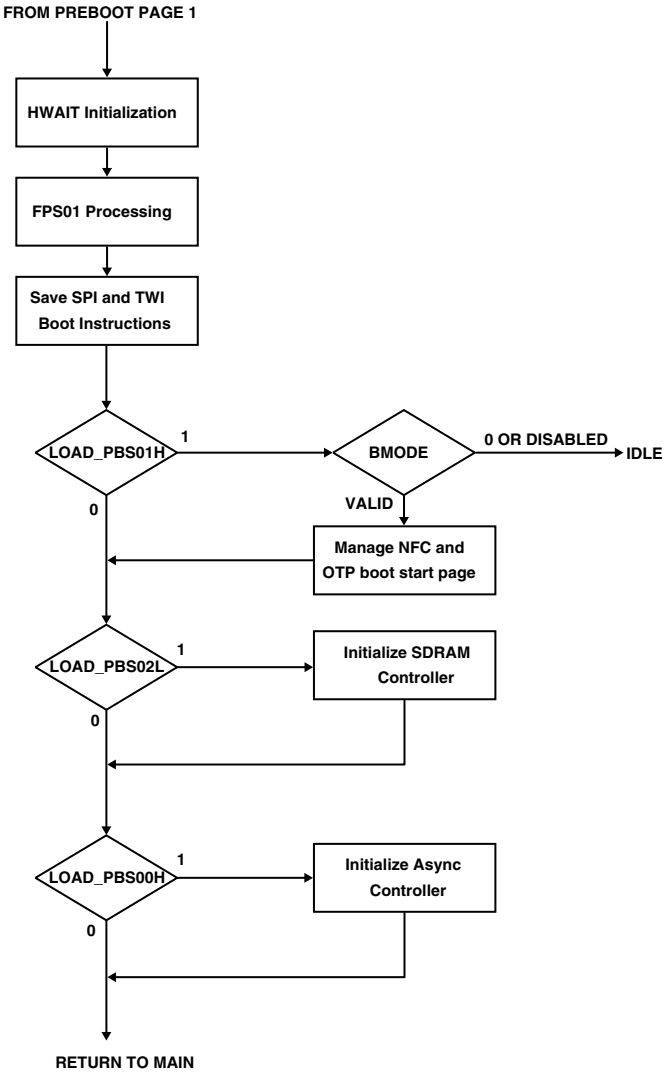


Figure 17-3. Preboot Flow 2 of 2

Factory Page Settings (FPS)

The content of the boot ROM is identical across all ADSP-BF54x processor Blackfin processors. The factory settings prevent the boot ROM from accidentally accessing resources that are not present on a given processor, which would result in unpredictable behavior and/or hardware errors. The boot kernel goes to a safe idle state when the user configures the `BMODE` pins to a boot mode that is not available on a specific part.

For this purpose, the preboot routine always reads the `FPS01L` and `FPS01H` half pages from OTP memory. These half pages contain factory trim values for the USB PHY controller that are managed at preboot time, as required. In addition, the `bfrom_SysControl()` routine reads the half page `FPS04H` and `FPS04L` to apply factory trim values to the voltage regulator and SDRAM controller.

Preboot Page Settings (PBS)

Four OTP pages optionally enable the user to customize the behavior of the processor immediately after reset. These four pages (eight half pages) can be seen as one contiguous pre-boot settings (PBS) block. By default, the block spans OTP pages 0x18 to 0x1B. The OTP pages serve the following purposes:

- PBS00L (by default, on half page 0x18L, see "Lower PBS00 Half Page" on page 17-110 for details)

  PLL and voltage regulator settings

  Boot customization

  Instruction whether to load further half pages

- PBS00H (by default, on half page 0x18H, see "Upper PBS00 Half Page" on page 17-114 for details)

Asynchronous EBIU register settings

- PBS01L (by default on half page 0x19L)

  Reserved

- PBS01H (by default, on half page 0x19H, see "Upper PBS01 Half Page" on page 17-116 for details)

  Disabling of boot modes

  NFC controller register settings

  OTP boot start page

- PBS02L (by default, on half page 0x1AL, see "Lower PBS02 Half Page" on page 17-118 for details)

  Synchronous EBIU register settings

- PBS02H (by default, on half page 0x1AH, see "Upper PBS02 Half Page" on page 17-119 for details)

  Synchronous EBIU register settings.

- PBS03L (by default, on half page 0x1BL, see "Reserved Half Pages" on page 17-120 for details)

  Reserved in current silicon revision. Do not use.

- PBS03H (by default, on half page 0x1BH, see "Reserved Half Pages" on page 17-120 for details)

  Reserved in current silicon revision. Do not use.

The preboot routine reads the main page `PBS00L` first. Since this page may instruct the preboot routine to alter the PLL settings, further pages may read more quickly. This page also instructs the preboot whether further OTP half pages have to be loaded and processed. By default, the `PBS00L` page reads all zeroes, and the preboot does not load further PBS pages.

Alternative PBS Pages

Especially during the development cycle, the user may fail to write the proper value to OTP memory and may make multiple attempts to get things right. Therefore, the PBS00L page provides a mechanism to invalidate the entire PBS block (consisting of pages (0x18, 0x19, 0x1A and 0x1B) and to use pages 0x1C to 0x1F instead. To do so, set the two OTP_INVALID bits (bits 62 and 63 on the PBS00L page). If both bits are set, the preboot routine disregards potential error codes returned by the bfrom_OtpRead() routine and continues processing from page 0x1C on. The active PBS block now spans the pages 0x1C to 0x1F. If the user wants to invalidate the second set of OTP pages as well, setting bits 62 and 63 on page 0x1C (which is the new PBS00L half page) instructs the preboot routine to continue at page 0x20, and so on.

Theoretically, this can be repeated up to page 0xD8L, if the pages are not required for other purposes. There are 49 chances to get things right, before a device may become useless. Note that every page that needs to be read by the preboot routine causes additional delay to the boot process.

Programming PBS Pages

Due to the need for error checking and correction (ECC), a 64-bit OTP half page must be written all at once. It is recommended that PBS pages be programmed only through the API function bfrom_OtpWrite().

If it is anticipated that the user is customizing the boot-related OTP pages for safety or security reasons, it is recommended that all PBS blocks be locked at production time to protect these pages from being tampered with in the field.

Reading OTP memory is subject to a potential failure rate. Since the preboot only accesses OTP memory through the bfrom_OtpRead() function, the ECC error correction is applied and the statistical failure rate is very low. However, the way the PBS00L page is tested for being invalid may at some point reduce the ECC

reliability. To keep failure rates at a minimum, it is a good idea to duplicate the content of pages 0x18–0x1B on pages 0x1C–0x1F. For production parts, the final block should be followed by its exact copy to maintain the lowest failure rates. Then, even the unlikely case where one of the OTP_INVALID bits is read incorrectly would not cause the boot to fail.

Recovering From Misprogrammed PBS Pages

The preboot mechanism provides a powerful method to customize the chip to the needs of the user. However, as a downside, there are chances that invalid values programmed to the PBS pages prevent the processor from operating within required operating conditions. There is specific risk when the PLL and the voltage regulator are programmed with meaningless values during the development cycle.

In such cases, the boot mode BMODE = b#0000 helps. In this mode, the preboot routine does not attempt to read any of the user-programmable PBS pages, and the boot kernel does not try to boot any data. Rather, the processor is idled immediately after the FPS pages have been processed. Using the in-circuit emulator, the user then has the option to invalidate the actual PBS settings by overwriting both OTP_INVALID bits in the actual PBS00L with 1s.

For safety reasons, none of the boot modes, except the emulator, can get control over the processor when in this state.

Customizing Power Management

When the processor awakes with default PLL and voltage regulator settings, the preboot mechanism can be used to alter these settings to custom values before the boot process takes place. This is done by programming the OTP half page PBS00L.

If the OTP_SET_PLL bit is programmed to a 1, the value in the OTP_PLL_DIV bit field is copied into the PLL_DIV register, and the OTP_PLL_CTL bit field is copied into the PLL_CTL register, followed by the required IDLE instruction (if the contents of PLL_CTL are being altered).

If the OTP_SET_VR bit is programmed to a 1, the value in the OTP_VR_CTL bit field is copied into the VR_CTL register, followed by the required IDLE instruction (if the contents of VR_CTL are being altered).

The preboot mechanism invokes the bfrom_SysControl() routine to alter the PLL and the voltage regulator. The bfrom_SysControl() routine not only performs custom instructions, it also applies correction values from factory OTP pages FPS01 and FPS04. See Chapter 18, "Dynamic Power Management" for details on the bfrom_SysControl() routine.

Customizing Booting Options

The OTP pages accessible by the preboot mechanism can also be used to customize some of the booting options. For example:

- TWI master boot mode operating frequency
- SPI master boot mode operating frequency
- SPI master boot mode read operation mode
- Start page for OTP boot mode
- HWAIT signal behavior
- Disabling of unwanted boot modes

In TWI master boot mode, the OTP_TWI_PRESCALE and OTP_TWI_CLKDIV values in the preboot half page PBS00L control the respective prescale and clock divider values written to the TWI0_CONTROL and TWI0_CLKDIV registers. The table of values can be found in "TWI Master Boot Mode" on

page 17-77. The bit field OTP_TWI_TYPE controls whether one, two, three or four address bytes are used to address the I^2C memory device. By default, two address bytes are used. The address bits embedded in the read command are not counted.

In SPI master boot mode, the OTP_SPI_BAUD register in the preboot half page PBS00L controls the value written to the SPI0_BAUD registers. By default, the clock divider value of 133 can be reduced in power-of-two steps. The table of values can be found in "SPI Master Boot Modes" on page 17-69. The OTP_SPI_BAUD bit instructs the boot kernel to use the 0x0B SPI read command instead of the normal 0x03 read command when accessing the SPI memory device.

In OTP boot mode, the boot kernel normally assumes that the boot stream starts at OTP page 0x40L. The user can change this start page by programming the OTP_START_PAGE bit field in the preboot half page PBS01H.

The boot host wait (HWAIT) signal is available in all boot modes. If the OTP_RESETOUT_HWAIT bit in the preboot half page PBS00L is set, the boot kernel does not toggle HWAIT. Rather, it simply drives it to simulate a reset output signal.

If the OTP_ALTERNATE_HWAIT bit in the same half page is set, the alternate GPIO pin (HWAITA) is used instead of HWAIT.

If safety or security of an application is impacted by the existence of certain boot modes, the boot mode disable bits in preboot half page PBS01H can be used to disable unwanted boot modes. If a disabled boot mode is chosen by the BMODE pins, the boot kernel goes into a safe idle state and stops processing. The half page PBS01H is only loaded when the OTP_LOAD_PBS01H bit in the PBS00L page is set.

Customizing the Asynchronous Port

The preboot half page PBS00H contains instructions to customize the asynchronous portion of the EBIU controller. This half page is only loaded and processed when the OTP_LOAD_PBS00H bit in the PBS00L half page is programmed to a 1.

The OTP_EBIU_AMG field is copied into the EBIU_AMGCTL register. While the lower bit controls the CLKOUT signal, the upper three AMBEN bits control which of the four asynchronous banks are enabled. For the FIFO boot mode, the three AMBEN bits are overruled and are all always set.

The preboot routine analyzes the three AMBEN bits and initializes the 16-bit portions (this routine is similar to the enabled banks in the EBIU_AMBCTL0 and EBIU_AMBCTL1 registers) with the value provided in the 16-bit OTP_EBIU_AMBCTL field. In this way, the bus timing of the asynchronous port can be customized prior to the boot process.

Half page PBS00H also contains the 16-bit OTP_EBIU_FCTL field which is copied directly to the EBIU_FCTL register.

Make sure that all bits in the OTP_EBIU_FCTL field that correspond with reserved bits in the EBIU_FCTL register are written with 0s.

The preboot routine ensures that a zero value is never written to the BCLK bit field.

The 8-bit value OTP_EBIU_MODE field is copied to the lower eight bits of the EBIU_MODE register. If any of the four memory banks has its BxMODE field set to a value of three, a device initialization sequence can be performed. All four banks are temporarily put into the asynchronous flash mode, and the four-bit OTP_EBIU_DEVSEQ field controls which sequence is performed. The 16-bit OTP_EBIU_DEVCFG word is part of the initialization sequence. Such a sequence is usually required to activate the bursting mode on multi-mode memories. Currently, the vendor-specific sequences shown in Table 17-3 are supported.

Table 17-3. Burst NOR Flash Initialization Sequences

OTP_EBIU_DEVSEQ = 2	OTP_EBIU_DEVSEQ = 4	OTP_EBIU_DEVSEQ = 6
Atmel, Intel, ST (16-bit)	Spansion (16-bit)	Samsung (16-bit)
w[OTP_EBIU_DEVCFG<<1] = 0x60	w[0x555<<1] = 0xAA	w[0x555<<1] = 0xAA
w[OTP_EBIU_DEVCFG<<1] = 0x03	w[0x2AA<<1] = 0x55	w[0x2AA<<1] = 0x55
w[0] = 0xFF	w[0x555<<1] = 0xD0	w[(OTP_EBIU_DEVCFG[6:0] <<12 | 0x555)<<1] = 0xC0
	w[0x000] = OTP_EBIU_DEVCFG	w[0x000] = 0xF0
	w[0x000] = 0xF0	

Whenever the PBS00H half page is processed, all EBIU signals that belong to the interface are enabled at the port muxing level. This includes the address pins on port H and port J, as well as the ARDY and bus request signals on port J. In flash boot mode, these signals are activated regardless of the OTP programming.

Finally, the 8-bit OTP_NFC_CTL field in the PBS01H half page initializes the eight least significant bits of the NFC_CTL register.

Customizing the Synchronous Port

Since many Blackfin applications require data and/or instruction code to be loaded into the SDRAM memory at boot time, the SDRAM controller must be initialized beforehand. This can be done by using either the "Initialization Code" on page 17-39 or the preboot mechanism described here. For the SDRAM boot mode, only the preboot mechanism is valid.

If the OTP_LOAD_PBS02L and OTP_LOAD_PBS02H bits in the PBS00L half page have been programmed to a 1, then the two preboot half pages PBS02L and PBS02H are also loaded and processed. These half pages initialize the SDRAM controller.

First, the preboot routine tests the SDRS bit in the EBIU_SDSTAT status register. To avoid reconfiguring an already enabled SDRAM controller, processing is bypassed if this bit is already set.

The lower twelve bits of the OTP_EBIU_SDRCC refresh rate value are written to the EBIU_SDRCC register. Similarly, the lower six bits of the OTP_EBIU_SDBCTL value are written to the EBIU_SDBCTL bank control register. The entire 32-bit OTP_EBIU_SDGCTL word is copied to the EBIU_SDGCTL global SDRAM control register.

To minimize access latencies during the boot process, an initial dummy access to the SDRAM is performed immediately after the SDRAM controller is set up. By default, a 32-bit dummy is read from address 0x0000 0000. If the OTP_EBIU_POWERON_DUMMY_WRITE bit is programmed to a "1", a 32-bit zero value is written to that address instead. This option takes less time but destroys the previous content of that memory location. Address 0x0000 0000 is rarely used for regular processing since it represents a target of NULL pointers.

Half page PBS02L contains the two 32-bit values OTP_EBIU_DDRCTL0 and OTP_EBIU_DDRCTL1 that are directly copied into the EBIU_DDRCTL0 and EBIU_DDRCTL1 SDRAM control registers, respectively.

Half page PBS02H contains the three 16-bit values OTP_EBIU_DDRCTL2L, OTP_EBIU_DDRCTL3L and OTP_EBIU_DDRQUEL that are copied into the lower 16 bits of the respective EBIU_DDRCTL2, EBIU_DDRCTL3 and EBIU_DDRQUE registers.

Basic Booting Process

Once the preboot routine returns, the boot kernel residing in the on-chip boot ROM starts processing the boot stream. The boot stream is either read from memory or received from a host processor. A boot stream represents the application data and is formatted in a special manner. The application data is segmented into multiple blocks of data. Each block begins with a block header. The header contains control words such as the destination address and data length information.

As Figure 17-4 on page 17-24 illustrates, the CCES or 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 *Loader and Utilities Manual* for information about the loader.

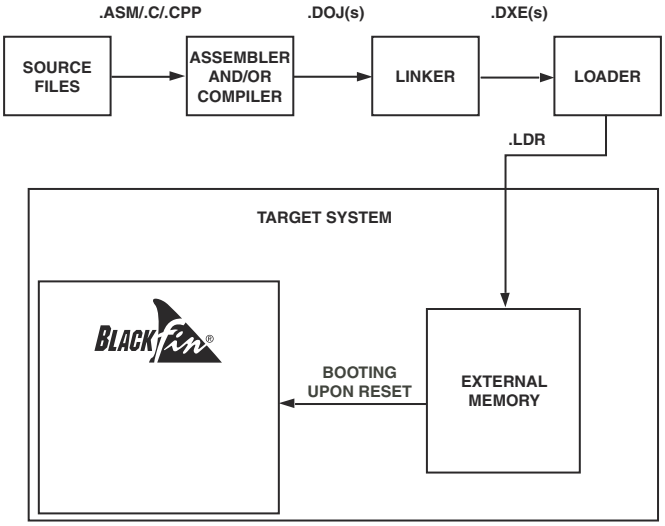


Figure 17-4. Project Flow for a Standalone System

Figure 17-5 on page 17-25 shows the parallel or serial boot stream contained in a flash memory device. In host boot scenarios, the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin processor. After reset, the headers are read and parsed by the on-chip boot ROM, and processed block-by-block. Payload data is copied to destination addresses, either in on-chip L1 and L2 memory or off-chip SRAM/SDRAM.

Booting into scratchpad memory (0xFFB0 0000–0xFFB0 0FFF) is not supported. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM. Similarly, booting into the upper 16 bytes of L1 data bank A (0xFF80 7FF0–0xFF80 7FFF by default) is not supported. These memory locations are used by the boot kernel for intermediate storage of block header information. These memory regions cannot be initialized at boot time. After booting, they can be used by the application during runtime.

When the `BFLAG_INDIRECT` flag for any block is set, as in TWI boot modes, the boot kernel uses another memory block in L1 data bank B (by default, 0xFF90 7E00–0xFF90 7FFF) for intermediate data storage. To avoid conflicts, the `elfloader` utility ensures this region is booted last.

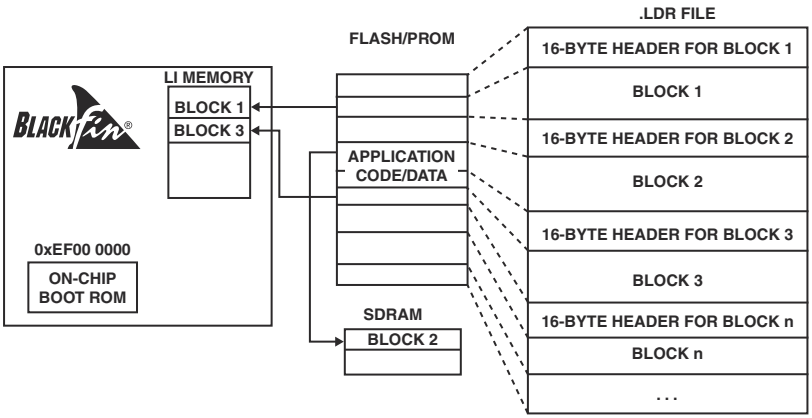


Figure 17-5. Booting Process

The entire source code of the boot ROM is shipped with the CCES or VisualDSP++ tools installation. Refer to the source code for any additional questions not covered in this manual. Note that minor maintenance work may be done to the content of the boot ROM when silicon is updated.

Block Headers

A boot stream consists of multiple boot blocks, as shown in Figure 17-5 on page 17-25. Every block is headed by a 16-byte block header. However, every block does not necessarily have a payload, as shown in Figure 17-6 on page 17-26.

The 16 bytes of the block header are functionally grouped into four 32-bit words, the BLOCK CODE, the TARGET ADDRESS, the BYTE COUNT, and the ARGUMENT fields.

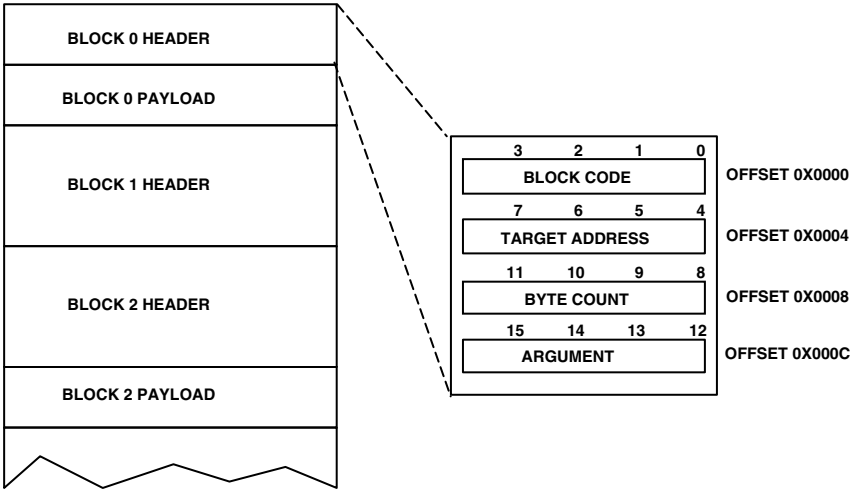


Figure 17-6. Boot Stream Headers

Block Code

The first 32-bit word is the BLOCK CODE. See Figure 17-7.

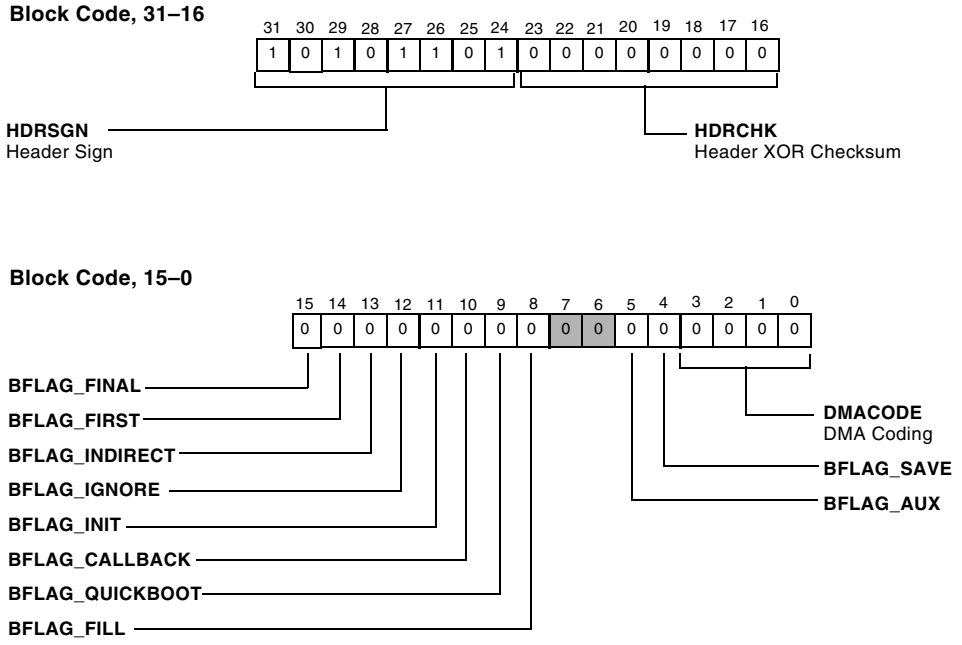


Figure 17-7. Block Code, 31–0

DMA Code Field

The DMA code (DMACODE) field instructs the boot kernel whether to use 8-bit, 16-bit or 32-bit DMA and how to program the source modifier of a memory DMA. Particularly in case of memory boot modes, this field is interrogated by the boot kernel to differentiate the 8-bit, 16-bit, and 32-bit cases.

The boot kernel tests this field only on the first block and ignores the field in further blocks (See Table 17-4).

Table 17-4. Bus and DMA Width Coding

DMA Code	DMA Width	Source DMA Modify	Application
0	reserved[1]		
1	8-bit	1	Default 8-bit boot from 8-bit source[2]
2	8-bit	2	Zero-padded 8-bit boot from 16-bit EBIU
3	8-bit	4	Zero-padded 8-bit boot from 32-bit EBIU[3]
4	8-bit	8	Zero-padded 8-bit boot from 64-bit EBIU[4]
5	8-bit	16	Zero-padded 8-bit boot from 128-bit EBIU[4]
6	16-bit	2	Default 16-bit boot from 16-bit source[5]
7	16-bit	4	Zero-padded 16-bit boot from 32-bit EBIU[3]
8	16-bit	8	Zero-padded 16-bit boot from 64-bit EBIU[4]
9	16-bit	16	Zero-padded 16-bit boot from 128-bit EBIU[4]
10	32-bit	4	Default 32-bit boot from 32-bit source[3, 5]
11	32-bit	8	Zero-padded 32-bit boot from 64-bit EBIU[4]
12	32-bit	16	Zero-padded 32-bit boot from 128-bit EBIU[4]
13	64-bit	8	Default 64-bit boot from 64-bit source[4]
14	64-bit	16	Zero-padded 64-bit boot from 128-bit EBIU[4]
15	128-bit	16	Default 128-bit boot from 128-bit source[4]

1 Reserved to differentiate from ADSP-BF53x boot streams.
2 Used by all byte-wise serial boot modes.
3 Applicable only to memory boot modes and OTP mode. This code is expected by OTP boot mode.
4 Not supported by ADSP-BF54x processor Blackfin products.
5 This is the only code supported by NAND flash boot.

Block Flags Field

Table 17-5. Block Flags

Bit	Name	Description
4	BFLAG_SAVE	Saves the memory of this block to off-chip memory in case of power failure or a hibernate request. This flag is not used by the on-chip boot kernel.
5	BFLAG_AUX	Nests special block types as required by special purpose second-stage loaders. This flag is not used by the on-chip boot kernel.
6	Reserved	
7	Reserved	
8	BFLAG_FILL	Tells the boot kernel to not process any payload data. Instead the target memory (specified by the `TARGET ADDRESS` and `BYTE COUNT` fields) is filled with the 32-bit value provided by the `ARGUMENT` word. The fill operation is always performed by 32-bit DMA; therefore target address and byte count must be divisible by four.
9	BFLAG_QUICKBOOT	Processes the block for full boot only. Does not process this block for a quick boot (warm boot).
10	BFLAG_CALLBACK	Calls a subfunction that may reside in on-chip or off-chip ROM or is loaded by an initcode in advance. Often used with the `BFLAG_INDIRECT` switch. If `BFLAG_CALLBACK` is set for any block, an initcode must register the callback function first. The function is called when either the entire block is loaded or the intermediate storage memory is full. The callback function can do advanced processing such as CRC checksum.
11	BFLAG_INIT	This flag causes the boot kernel to issue a `CALL` instruction to the target address of the boot block after the entire block is loaded. The initcode should return by an `RTS` instruction. It may or may not be overwritten by application data later in the boot process. If the code is loaded earlier or resides in ROM, the init block can be zero sized (no payload).

Table 17-5. Block Flags (Cont'd)

Bit	Name	Description
12	BFLAG_IGNORE	Indicates a block that is not booted into memory. It instructs the boot kernel to skip the number of bytes of the boot stream as specified by BYTE COUNT. In master boot modes, the boot kernel simply modifies its source address pointer. In this case the BYTE COUNT value can be seen as a 32-bit two's-complement offset value to be added to the source address pointer. In slave boot modes, the boot kernel actively loads and changes the payload of the block. In slave modes the byte count must be a positive value.
13	BFLAG_INDIRECT	Boots to an intermediate storage place, allowing for calling an optional callback function, before booting to the destination. This flag is used when the boot source does not have DMA support (TWI for example) and either the destination cannot be accessed by the core (L1 instruction SRAM) or cannot be efficiently accessed by the core (SDRAM or RAM). This flag is also used when CALLBACK requires access to data to calculate a checksum, or when performing tasks such as decryption or decompression.
14	BFLAG_FIRST	This flag, which is only set on the first block of a DXE, tells the boot kernel about the special nature of the TARGET ADDRESS and the ARGUMENT fields. The TARGET ADDRESS field holds the start address of the application. The ARGUMENT field holds the offset to the next DXE.
15	BFLAG_FINAL	This flag causes the boot kernel to pass control over to the application after the final block is processed. This flag is usually set on the last block of a DXE unless multiple DXEs are merged.

The BFLAG_FIRST flag must not be combined with the BFLAG_FILL flag. The BFLAG_FIRST flag may be combined with the BFLAG_IGNORE flag to deposit special user data at the top of the boot stream. Note the special importance of the elfloader -readall switch.

Header Checksum Field

The header checksum (HDRCHK) field holds a simple XOR checksum of the other 31 bytes in the boot block header. The boot kernel jumps to the error routine if the result of an XOR operation across all 32 header bytes

(including the HDRCHK value) differs from zero. The default error routine is a simple IDLE; instruction. The user can overwrite the default error handler using the initcode mechanism.

Header Sign Field

The header signature (HDRSGN) byte always reads as 0xAD and is used to verify whether the block pointer actually points to a valid block. The HDRSGN byte can also be used as a boot stream version control. For the ADSP-BF54x, ADSP-BF52x and ADSP-BF51x Blackfin processors, the byte always reads 0xAD. The ADSP-BF53x boot streams always read 0xFF. The ADSP-BF561 boot streams always read 0xA0.

Target Address

This 32-bit field holds the target address where the boot kernel loads the block payload data. When the BFLAG_FILL flag is set, the boot kernel fills the memory with the value stored in the ARGUMENT field starting at this address. If the BFLAG_INIT flag is set the kernel issues a CALL(TARGET ADDRESS) instruction after the optional payload is loaded.

If the BFLAG_FIRST flag is set, the TARGET ADDRESS field contains the start address of the application to which the boot kernel jumps at the end of the boot process. This address will also be stored in the EVT1 register. The elf-loader utility sets this value to 0xFFA0 0000 for compatibility with other Blackfin products.

The target address should be divisible by four, because the boot kernel uses 32-bit DMA for certain operations. The target address must point to valid on-chip or off-chip memory locations. When booting to external memories, the memory controller must first be set up by either the pre-boot or the initcode mechanism. When booting through peripherals that do not support DMA transfers, such as the TWI or OTP boot mode, the BFLAG_INDIRECT flag must be set if the target address points to L1 instruction memory. For performance reasons this is also recommended when booting to off-chip memories.

For the TWI or OTP boot modes, the elfloader utility manages the BFLAG_INDIRECT flag automatically. Refer to *Loader and Utilities Manual* for manual control of the flag.

Booting to scratchpad memory is not supported. The scratchpad memory functions as a stack for the boot kernel. The L1 data memory locations 0xFF80 7FF0 to 0xFF80 7FFF are used by the boot kernel and should not be overwritten by the application. The memory range used for intermediate storage as controlled by the BFLAG_INDIRECT switch should only be booted after the last BFLAG_INDIRECT bit is processed. By default the address range 0xFF90 7E00–0xFF90 7FFF is used for intermediate storage.

For normal boot operation, the target address points to RAM memory. There are however a few exceptions where the target address can point to on-chip or off-chip ROM. For example a zero-sized BFLAG_INIT block would instruct the boot kernel to call a subroutine residing in ROM or flash memory. This method is used to activate the CRC32 feature.

Byte Count

This 32-bit field tells the boot kernel how many bytes to process. Normally, this is the size of the payload data of a boot block. If the BFLAG_FILL flag is set there is no payload. In this case the BYTE COUNT field uses the value in its ARGUMENT field to tell the boot kernel how many bytes to process.

The byte count is a 32-bit value that should be divisible by four. Zero values are allowed in all block types. Most boot modes are based upon DMA operation which are only 16-bit words for Blackfin processors. The boot kernel may therefore start multiple DMA work units for large boot blocks. This enables a single block to fill to zero the entire SDRAM memory, for example, resulting in compact boot streams. The HWAIT signal may toggle for each work unit.

If the BFLAG_IGNORE flag is set, the byte count is used to redirect the boot source pointer to another memory location. In master boot modes, the byte count is a two's-complement (signed long integer) value. In slave boot modes, the value must be positive.

Argument

This 32-bit field is a user variable for most block types. The value is accessible by the initcode or the callback routine and can therefore be used for optional instructions to these routines. When the CRC32 feature is activated, the ARGUMENT field holds the checksum over the payload of the block.

When the BFLAG_FILL flag is set there is no payload. The argument contains the 32-bit fill value, which is most likely a zero.

If the BFLAG_FIRST flag is set, the argument contains the relative next-DXE pointer for multi-DXE applications. For single-DXE applications the field points to the next free boot source address after the current DXE's boot stream.

Boot Host Wait (HWAIT) Feedback Strobe

The HWAIT feedback strobe is a handshake signal that is used to hold off the host device from sending further data while the boot kernel is busy.

On ADSP-BF54x processors this feature is implemented by a GPIO that is toggled by the boot kernel as required. By default the PB11 GPIO is used for this purpose. If the OTP_ALTERNATE_HWAIT fuse in OTP memory page PBS00L is programmed, the boot kernel uses the PH7 GPIO instead.

The signal polarity of the HWAIT strobe is programmable by an external resistor in the 10 kΩ range.

A pull-up resistor instructs the HWAIT signal to be active high. In this case the host is permitted to send header and footer data when HWAIT is low, but should pause while HWAIT is high. This is the mode used in SPI slave boot on other Blackfin products.

Similarly, a pull-down resistor programs active-low behavior.

Note that the HWAIT signal is implemented slightly differently than on ADSP-BF53x Blackfin processors. In the ADSP-BF54x processor processors, the meaning of the pulling resistor is inverted and HWAIT is asserted by default during reset and preboot.

After preboot, the boot kernel first senses the polarity on the respective HWAIT pin. Then it enables the output driver but keeps the signal in its asserted state. The signal is not released until the boot kernel is ready for data, or when a receive DMA is started. As soon as the DMA completes, HWAIT becomes active again.

The boot host wait signal holds the host from booting in any slave boot mode and prevents it from being overrun with data. The HWAIT signal is, however, available in all boot modes with the exception of the NAND flash boot mode. In some cases it is redundant to other handshake mechanisms, such as the UART $\overline{\text{RTS}}$ signal. See "UART Slave Mode Boot" on page 17-82.

In general the host device must interrogate the HWAIT signal before every word that is sent. This requirement can be relaxed for boot modes using on-chip peripherals that feature larger receive FIFOs. However, the host must not rely on the DMA FIFO since its content is cleared at the end of a DMA work unit.

While the HWAIT signal is only used for boot purposes, it may also play a significant role after booting. In slave boot modes, for example, the host device does not necessarily know whether the Blackfin processor is in an active mode or a power-down mode. For example, the HWAIT signal can be used to signal when the processor is in hibernate mode.

Using HWAIT as Reset Indicator

While the HWAIT signal is mandatory in some boot modes, it is optional in others. When not required for booting, the behavior of the HWAIT signal (or alternate HWAITA signal, see on page 17-19) can be changed by programming the OTP_RESETOUT_HWAIT bit in OTP page PBS00L.

If this bit is set, HWAIT does not toggle during the boot process. Rather, after page PBS00L is processed (and therefore the PLL has settled) the preboot routine first enables the HWAIT GPIO as an input and senses its state. Then HWAIT becomes an output and is driven to the invert of the state that is sensed. An external pulling resistor is required. If using a pull-up resistor, the HWAIT signal is driven low for the rest of the boot process (and beyond). If using a pull-down resistor, HWAIT is driven high.

With a pull-down resistor, this feature can be used to simulate an active-low reset output. When the processor is reset, or in hibernate, the GPIO is in a high impedance state and HWAIT is pulled low by the resistor. As soon as the processor recovers and has settled the PLL again, the HWAIT is driven high and can alert external circuits.

Boot Termination

After the successful download of the application into the bootable memory, the boot kernel passes control to the user application. By default this is performed by jumping to the vector stored in the EVT1 register. The boot kernel provides options to execute an RTS instruction or a RAISE 1 instruction instead. The default behavior can be changed by an initcode routine. The EVT1 register is updated by the boot kernel when processing the BFLAG_FIRST block. See “Servicing Reset Interrupts” on page 17-10 to learn how the application can take control.

Before the boot kernel passes program control to the application it does some housekeeping. Most of the registers that were used are changed back to their default state but some register values may differ for individual boot modes. DMA configuration registers and primary register control

registers (UART0_LCR, SPI0_CTL, HOST_CONTROL, etc.) are restored, while others are purposely not restored. For example SPI0_BAUD, UART0_DLH and UART_DLL remain unchanged so that settings obtained during the booting process are not lost.

Single Block Boot Streams

The simplest boot stream consists of a single block header and one contiguous block of instructions. With appropriate flag instructions the boot kernel loads the block to the target address and immediately terminates by executing the loaded block.

Table 17-6 shows an example of a single block boot stream header that could be loaded from any serial boot mode. It places a 256-byte block of instructions at L1 instruction SRAM address 0xFFA0 0000. The flags BFLAG_FIRST and BFLAG_FINAL are both set at the same time. Advanced flags, such as BFLAG_IGNORE, BFLAG_INIT, BFLAG_CALLBACK and BFLAG_FILL, do not make sense in this context and should not be used.

Table 17-6. Header for a Single Block Boot Stream

Field	Value	Comments
BLOCK CODE	0xAD33 C001	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | (DMACODE & 0x1)
TARGET ADDRESS	0xFFA0 0000	Start address of block and application code
BYTE COUNT	0x0000 0100	256 bytes of code
ARGUMENT	0x0000 0100	Functions as next-DXE pointer in multi-DXE boot streams

With the BFLAG_FIRST flag set, the ARGUMENT field functions as the next-DXE pointer. This is a relative pointer to the next free source address or to the next DXE start address in a multi-DXE stream.

Direct Code Execution

Applications may want to avoid long booting times and start code execution directly from 16-bit flash or SDRAM memory. This feature is called direct code execution. This is a special case of boot termination that replaces the no-boot/bypass mode in the ADSP-BF53x Blackfin processors.

An initial boot block header is needed for the processor to fetch and execute program code from the boot device as early as possible. The safety mechanisms of the block, such as the header signature and the XOR checksum, avoid unpredictable processor behavior due to the boot memory not being programmed with valid data yet. Rather than blindly executing code, the boot kernel first executes the preboot routine for system customization, then loads the first block header and checks it for consistency. If the block header is corrupted, the boot kernel goes into a safe idle state and does not start code execution.

If the initial block header checks good, the boot kernel interrogates the block flags. If the block has the BFLAG_FINAL flag set, the boot kernel immediately terminates and jumps directly to the address stored in the EVT1 register. To cause the boot kernel to customize the EVT1 register in advance, the initial blocks must also have the BFLAG_FIRST flag set. The TARGET ADDRESS field is then copied to the EVT1 register. In this way, the TARGET ADDRESS field of the initial block defines the start address of the application.

For example in BMODE = 0001, when the block header described in Table 17-7 on page 17-38 is placed at address 0x2000 0000, the boot kernel is instructed to issue a JUMP command to address 0x2000 0020.

The development tools must be instructed to link the above block to address 0x2000 0000 and the application code to address 0x2000 0020. An example shown in "Direct Code Execution" on page 17-155 illustrates how this is accomplished using the CCES or VisualDSP++ tools suite.

Table 17-7. Initial Header for Direct Code Execution in BMODE = 0001

Field	Value	Comments
BLOCK CODE	0xAD7B D006	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | BFLAG_IGNORE | (DMACODE & 0x6)
TARGET ADDRESS	0x2000 0020	Start address of application code
BYTE COUNT	0x0000 0010	Ignores 16 bytes to provide space for control data such as version code and build data. This is optional and can be zero.
ARGUMENT	0x0000 0010	Functions as next-DXE pointer in multi-DXE boot streams

Similarly for direct code execution in the SDRAM boot mode (BMODE = 1010), an initial block as shown in Table 17-8 has to be linked to address 0x0000 0010.

Table 17-8. Initial Header for Direct Code Execution in BMODE = 1010

Field	Value	Comments
BLOCK CODE	0xAD5B D006	0xAD000000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | BFLAG_IGNORE | (DMACODE & 0x6)
TARGET ADDRESS	0x0000 0020	Start address of application code
BYTE COUNT	0x0000 0000	No bubble for control data
ARGUMENT	0x0000 0000	Functions as next-DXE pointer in multi-DXE boot streams

For multi-DXE boot streams, Figure 17-11 on page 17-59 shows a linked list of initial blocks that represent different applications.

Advanced Boot Techniques

The following sections describe advanced boot techniques.

Initialization Code

Initcode routines are subroutines that the boot kernel calls during the booting process. The user can customize and speed up the booting mechanisms using this feature. Traditionally, an initcode is used to set up system PLL, bit rates, wait states and the SDRAM controller. If executed early in the boot process, the boot time can be significantly reduced.

After the payload data is loaded for a specific boot block, if the `BFLAG_INIT` flag is set, the boot kernel issues a `CALL` instruction to the target address of the block.

On ADSP-BF54x processor Blackfin processors, initcode routines follow the C language calling convention so they can be coded in C language or assembly.

The expected prototype is
`void initcode(ADI_BOOT_DATA* pBootStruct);`

The header files define the `ADI_BOOT_INITCODE_FUNC` type: `typedef void ADI_BOOT_INITCODE_FUNC (ADI_BOOT_DATA* ) ;`

Optionally, the initcode routine can interrogate the formatting structure and customize its own behavior or even manipulate the regular boot process. A pointer to the structure is passed in the `R0` register. Assembly coders must ensure that the routine returns to the boot kernel by a terminating `RTS` instruction.

Initcodes can rely on the validity of the stack, which resides in scratchpad memory. The `ADI_BOOT_DATA` structure resides on the stack. Rules for register usage conform to the compiler conventions. See *C/C++ Compiler and Library Manual for Blackfin Processors* for more information.

In the simple case, initcodes consist of a single instruction section and are represented by a single block within the boot stream. This block has the BFLAG_INIT bit set.

An init block can consist of multiple sections where multiple boot blocks represent the initcode within the boot stream. Only the last block has the BFLAG_INIT bit set.

The elfloader utility ensures that the last of these blocks vectors to the initcode entry address. The utility instructs the on-chip boot ROM to execute a CALL instruction to the given target address.

When the on-chip boot ROM detects a block with the BFLAG_INIT bit set, it boots the block into Blackfin memory and then executes it by issuing a CALL to its target address. For this reason, every initcode must be terminated by an RTS instruction to ensure that the processor vectors back to the on-chip boot ROM for the rest of the boot process.

Sometimes initcode boot blocks have no payload and the BYTE COUNT field is set to zero. Then the only purpose of the block may be to instruct the boot kernel to issue the CALL instruction.

Initcode routines can be very different in nature. They might reside in ROM or SRAM. They might be called once during the booting process or multiple times. They might be volatile and be overwritten by other boot blocks after executing, or they might be permanently available after boot time. The boot kernel has no knowledge of the nature of initcodes and has no restrictions in this regard. Refer to *Loader and Utilities Manual* for how this feature is supported by the tools chain.

It is the user's responsibility to ensure that all code and data sections that are required by the initcode are present in memory by the time the initcode executes. Special attention is required if initcodes are written in C or C++ language. Ensure that the initcode does not contain calls to the runtime libraries. Do not assume that parts of the runtime environment, such as the heap are fully functional. Ensure that all runtime components are loaded and initialized before the initcode executes.

The elfloader utility provides two different mechanisms to support the initcode feature.

- The `-init initcode.dxe` command line switch
- The `-initcall address/symbol` command line switch

If enabled by the elfloader `-init initcode.dxe` command-line switch, the initcode is added to the beginning of the boot stream. Here, `initcode.dxe` refers to the user-provided custom initialization executable— a separate project. Figure 17-8 on page 17-42 shows a boot stream example that performs the following steps.

1. Boot initcode into L1 memory.
2. Execute initcode.
3. Initcode initializes the SDRAM controller and returns.
4. Overwrite initcode with final application code.
5. Boot data/code into SDRAM.
6. Continue program execution with block n.

Although `initcode.dxe` files are built as CrossCore Embedded Studio or VisualDSP++ projects, they differ from standard projects. Initcodes provide only a callable sub-function, so they look more like a library than an application. Nevertheless, unlike library files (`.DLB` file extension), the symbol addresses have already been resolved by the linker.

An initcode is always a heading for the regular application code. Consequently whether the initcode consists of one or multiple blocks, it is not terminated by a `BFLAG_FINAL` bit indicator—this would cause the boot ROM to terminate the boot process.

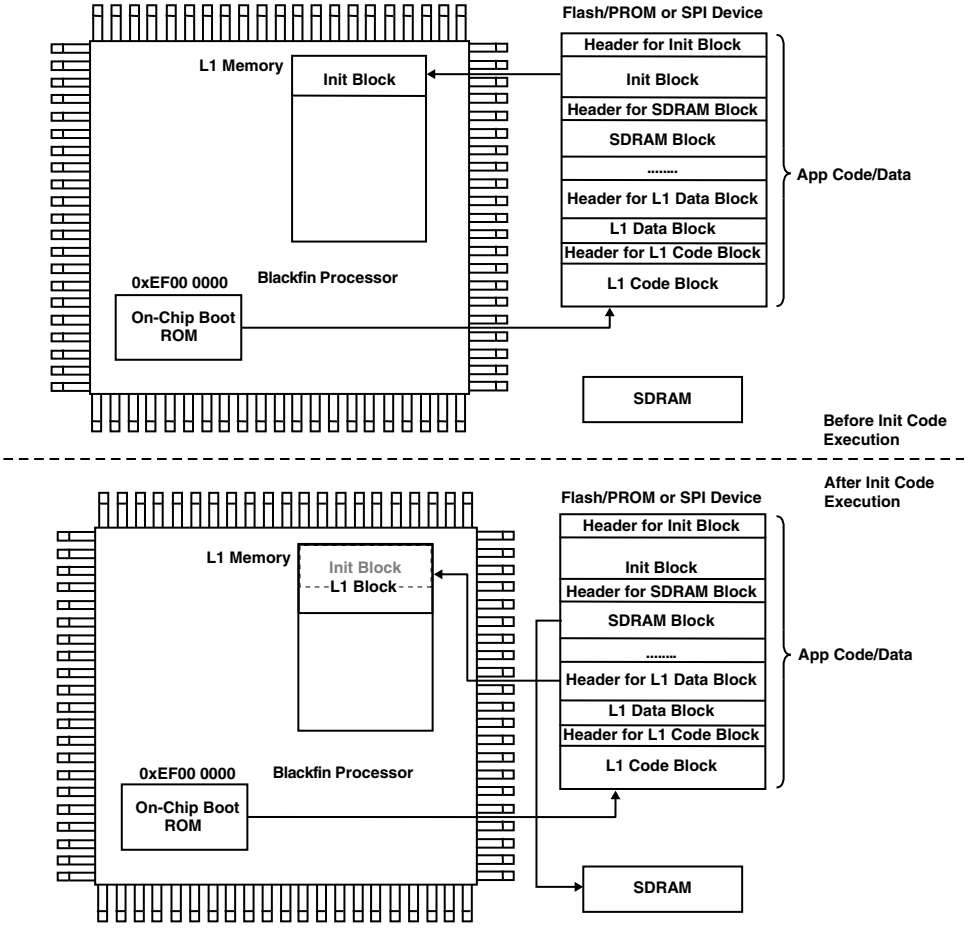


Figure 17-8. Initialization Code Execution/Boot

It is advantageous to have a clear separation between the initcode and the application by using the `-init` switch. If this separation is not needed, the elfloader `-initcall` command-line switch might be preferred.
It enables fractions of the application code to be traded as initcode during the boot process. See *Loader and Utilities Manual* for further details.

Initcode examples are shown in "Programming Examples" on page 17-145.

Quick Boot

In some booting scenarios, not all memories need to be re-initialized. For example in a wake-up from hibernate state, off-chip SRAM might not be impacted if it was powered while the processor was in hibernate state. Dynamic RAM might also not be impacted if it was put into self-refresh mode before the processor powered down.

The ADSP-BF54x processor processor's boot kernel can conditionally process boot blocks. The normal scenario is all boot, the shortened version is quick boot. It relies on the following primitives.

- The `SYSCR` register is read to determine what kind of boot is expected from the boot kernel. Refer to Figure 17-40 on page 17-106.

  The `WURESET` bit is used to distinguish between cold boot and warm boot situations and to identify wake-up from hibernate situations.

  The `BCODE` bit field in the `SYSCR` register can overrule the native decision of the boot kernel for a software boot. See the flowchart in Figure 17-1 on page 17-9.

- The `BFLAG_WAKEUP` bit in the `dFlag` word of the `ADI_BOOT_DATA` structure indicates that the final decision was to perform a quick boot. If the boot kernel is called from the application, then the application can control the boot kernel behavior by setting the `BFLAG_WAKEUP` flag accordingly. See the `dFlags` variable on Figure 17-54 on page 17-125.

- The `BFLAG_QUICKBOOT` flag in the `BLOCK CODE` word of the block header controls whether the current block is ignored for quick boot.

If both the global BFLAG_WAKEUP and the block-specific BFLAG_QUICKBOOT flags are set, the boot kernel ignores those blocks. But since the BFLAG_INIT, BFLAG_CALLBACK, BFLAG_FINAL, and BFLAG_AUX flags are internally cleared and the BFLAG_IGNORE flag is toggled, through double negation, the "ignore the ignore block" command instructs the boot kernel to process the block.

Although the BFLAG_INIT flag is suppressed in quick boot, the user may not want to combine the BFLAG_INIT flag with the BFLAG_QUICKBOOT flag. The initialization code can interrogate the BFLAG_WAKEUP flag and execute conditional instructions. For more information see "Quickboot With Restore From SDRAM" on page 17-152.

Indirect Booting

The processor's boot kernel provides a control mechanism to let blocks either boot directly to their final destination or load to an intermediate storage place, then copy the data to the final destination in a second step. This feature is motivated by the following requirements.

- Some boot modes such as TWI do not use DMA. They load data by core instruction. The core cannot access some memories directly (for example L1 instruction SRAM), or is less efficient than the DMA in accessing some memories (for example, external SDRAM).
- In some advanced booting scenarios, the core needs to access the boot data during the booting process, for example in processing decompression, decryption and checksum algorithms at boot time. The indirect booting option helps speed-up and simplify such scenarios. Software accesses off-chip memory less efficiently and cannot access data directly if it resides in L1 instruction SRAM.

Indirect booting is not a global setting. Every boot block can control its own processing by the BFLAG_INDIRECT flag in the block header.

In general a boot block may not fit into the temporary storage memory so the boot kernel processes the block in multiple steps. The larger the temporary buffer, the faster the boot process. By default the L1 data memory region between 0xFF90 7E00 and 0xFF90 7FFF is used for intermediate storage. Initialization code can alter this region by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure. The default region is at the upper end of a physical memory block. When increasing the `dTempByteCount` value, `pTempBuffer` also has to change.

Callback Routines

Callback routines, like initialization codes, are user-defined subroutines called by the boot kernel at boot time. The `BFLAG_CALLBACK` flag in the block header controls whether the callback routine is called for a specific block.

There are several differences between initcodes and callback routines. While the `BFLAG_INIT` flag causes the boot kernel to issue a `CALL` instruction to the target address of the specific boot block, the `BFLAG_CALLBACK` flag causes the boot kernel to issue a `CALL` instruction to the address held by the `pCallBackFunction` pointer in the `ADI_BOOT_DATA` structure. While a boot stream can have multiple individual initcodes, it can have just one callback routine. In the standard boot scenario, the callback routine has to be registered by an initcode prior to the first block that has the `BFLAG_CALLBACK` flag set.

The purpose of the callback routine is to apply standard processing to the block data. Typically, callback routines contain checksum, decryption, decompression, or hash algorithms. Checksum or hash words can be passed through the block header `ARGUMENT` field.

Since callback routines require access to the payload data of the boot blocks, the block data must be loaded before it can be processed. Unlike initcodes, a callback usually resides permanently in memory. If the block

is loaded to L1 instruction memory or off-chip memory, the BFLAG_CALLBACK flag is likely combined with the BFLAG_INDIRECT bit. The boot kernel performs these steps in the following order.

1. Data is loaded into the temporary buffer defined by the pTempBuffer variable.
2. The CALL to the pCallBackFunction is issued.
3. After the callback routine returns, the memory DMA copies data to the destination.

If a block does not fit into the temporary buffer, for example when the BLOCK COUNT is greater than the dTempByteCount variable, the three steps are executed multiple times until all payload data is loaded and processed. The boot kernel passes the parameter dCbFlags to the callback routine to tell it that it is being invoked the first or the last time for a specific block. To store intermediate results across multiple calls the callback routine can use the uwUserShort and dUserLong variables in the ADI_BOOT_DATA structure.

Callback routines meet C language calling conventions for subroutines. The prototype is as follows.

```
s32 CallBackFunction (ADI_BOOT_DATA* pBootStruct,
ADI_BOOT_BUFFER* pCallbackStruct, s32 dCbFlags);
```

The header file defines the ADI_BOOT_CALLBACK_FUNC type the following way:

```
typedef s32 ADI_BOOT_CALLBACK_FUNC (ADI_BOOT_DATA*,
ADI_BOOT_BUFFER*, s32 ) ;
```

The pBootStruct argument is passed in R0 and points to the ADI_BOOT_DATA structure used by the boot kernel. These are handled by the pTempBuffer and dTempByteCount variables as well as the pHeader pointer to the ARGUMENT field. The callback routine may process the block further by modifying the pTempBuffer and dTempByteCount variables.

The pCallbackStruct structure passed in R1 provides the address and length of the data buffer. When the BFLAG_INDIRECT flag is not set, the pCallbackStruct contains the target address and byte count of the boot block. If the BFLAG_INDIRECT flag is set, the pCallbackStruct contains a copy of the pTempBuffer. Depending on the size of the boot block and processing progress, the byte count provided by pCallbackStruct equals either dTempByteCount or the remainder of the byte count.

When the BFLAG_INDIRECT flag is set along with the BFLAG_CALLBACK flag, memory DMA is invoked by the boot kernel after the callback routine returns. This memory DMA relies on the pCallbackStruct structure not the global pTempBuffer and dTempByteCount variables.

The callback routine can control the source of the memory DMA by altering the content of the pCallbackStruct structure, as may be required if the callback routine performs data manipulation such as decompression.

The dCbFlags parameter passed in R2 tells the callback routine whether it is invoked the first time (CBFLAG_FIRST) or whether it is called the last time (CBFLAG_FINAL) for a specific block. The CBFLAG_DIRECT flag indicates that the BFLAG_INDIRECT bit is not active so that the callback routine will only be called once per block. When the CBFLAG_DIRECT flag is set, the CBFLAG_FIRST and CBFLAG_FINAL flags are also set.

```
#define CBFLAG_FINAL        0x0008
#define CBFLAG_FIRST        0x0004
#define CBFLAG_DIRECT       0x0001
```

A callback routine also has a boolean return parameter in register R0. If the return value is non-zero, the subsequent memory DMA does not execute. When the CBFLAG_DIRECT flag is set, the return value has no effect.

Error Handler

While the default handler simply puts the processor into idle mode, an initcode routine can overwrite this pointer to create a customized error handler. The expected prototype is

```
void ErrorFunction (ADI_BOOT_DATA* pBootStruct, void
*pFailingAddress);
```

Use an initcode to write the entry address of the error routine to the `pErrorFunction` pointer in the `ADI_BOOT_DATA` structure. The error handler has access to the boot structure and receives the instruction address that triggered the error.

CRC Checksum Calculation

The ADSP-BF54x processor Blackfin processors provide an initcode and a callback routine in ROM that can be used for CRC32 checksum generation during boot time. The checksum routine only verifies the payload data of the blocks. The block headers are already protected by the native XOR checksum mechanism.

Before boot blocks can be tagged with the `BFLAG_CALLBACK` flag to enable checksum calculation on the blocks, the boot stream must contain an initcode block with no payload data and with the CRC32 polynomial in the block header `ARGUMENT` word.

The initcode registers a proper CRC32 wrapper to the `pCallBackFunction` pointer. The registration principle is similar to the XOR checksum example shown in “Programming Examples” on page 17-145.

Load Functions

With the exception of the Host DMA boot modes, all boot modes are processed by a common boot kernel algorithm. The major customization is done by a subroutine that must be registered to the `pLoadFunction` pointer in the `ADI_BOOT_DATA` structure. Its simple prototype is as follows.

```
void LoadFunction (ADI_BOOT_DATA* pBootStruct);
```

The header files define the following type:

```
typedef void ADI_BOOT_LOAD_FUNC (ADI_BOOT_DATA* ) ;
```

For a few scenarios some of the flags in the `dFlags` word of the `ADI_BOOT_DATA` structure, such as `BFLAG_PERIPHERAL` and `BFLAG_SLAVE`, slightly modify the boot kernel algorithm.

The boot ROM contains several load functions. One performs a memory DMA for flash boot, others perform peripheral DMAs or load data from booting source by polling operation. The first is reused for fill operation and indirect booting as well.

In second-stage boot schemes, the user can create customized load functions or reuse the original `BFROM_PDMA` routine and modify the `pDmaControlRegister`, `pControlRegister` and `dControlValue` values in the `ADI_BOOT_DATA` structure. The `pDmaControlRegister` points to the `DMAx_CONFIG` or `MDMA_Dx_CONFIG` register. When the `BFLAG_SLAVE` flag is not set, the `pControlRegister` and `dControlValue` variables instruct the peripheral DMA routine to write the control value to the control register every time the DMA is started.

Load functions written by users must meet the following requirements.

- Protect against `dByteCount` values of zero.
- Multiple DMA work units are required if the `dByteCount` value is greater than 65536.
- The `pSource` and `pDestination` pointers must be properly updated.

In slave boot modes, the boot kernel uses the address of the `dArgument` field in the `pHeader` block as the destination for the required dummy DMAs when payload data is consumed from `BFLAG_IGNORE` blocks. If the load function requires access to the block's `ARGUMENT` word, it should be read early in the function.

The most useful load functions `BFROM_MDMA` and `BFROM_PDMA` are accessible through the jump table. Others, do not have entries in the jump table. Their start address can be determined with the help of the hook routine when calling the respective `BFROM_SPIBOOT`, `BFROM_OTPBOOT` etc. functions. In this way they can be repurposed for runtime utilization.

Calling the Boot Kernel at Runtime

The boot kernel's primary purpose is to boot data to memory after power-up and reset cycles. However some of the routines used by the boot kernel might be of general value to the application. The boot ROM supports reuse of these routines as C-callable subroutines. Programs such as second-stage boot kernels, boot managers, and firmware update tools may call the function in the ROM at runtime. This could load entirely different applications or a fraction of an application, such as a code overlay or a coefficient array.

To call these boot kernel subroutines, the boot ROM provides an API at address 0xEF00 0000 in the form of a jump table.

When calling functions in the boot ROM, the user must ensure the presence of a valid stack following C language conventions. See *C/C++ Compiler and Library Manual for Blackfin Processors* for details.

Debugging the Boot Process

If the boot process fails, very little information can be gained by watching the chip from outside. In master boot modes, the interface signals can be observed. In slave boot modes only the HWAIT (or the $\overline{\text{RTS}}$) signal tells about the progress of the boot process.

However, by using the emulator, there are many possibilities for debugging the boot process. The entire source code of the boot kernel is provided with the CCES or VisualDSP++ installation. This includes the project executable (DXE) file. The LOAD SYMBOLS feature helps to navigate the program. Note that the content of the ROM might differ between silicon revisions. Hardware breakpoints and single-stepping capabilities are also available. Since the content of the L1 instruction ROM cannot be read out by the emulator (as this ROM is not supported by the ITEST feature), these instructions are not displayed in the disassembly window.

Table 17-9 shows program symbols that are of interest.

Table 17-9. Boot Kernel Symbols for Debug

Symbol	Comment
_bootrom.assert.default	If the program counter halts at the IDLE instruction at the _bootrom.assert.default address, either the boot kernel or the preboot has detected an error condition and will not continue the boot process. A misformatted boot stream, checksum errors, or invalid PBS settings are the most likely causes of such an error. The RETS register points to the failing routine. When stepping a couple of instructions further, there is a way to ignore the error and to continue the boot process by clearing the >ASTAT register while the emulator steps over the subsequent IF CC JUMP 0 instruction.
_bootrom.bootmenu	If the emulator hits a hardware breakpoint at the _bootrom.bootmenu address, this indicates that the preboot returned properly. Otherwise the program may hang during preboot due to improper PBS settings or invalid boot modes.

Table 17-9. Boot Kernel Symbols for Debug (Cont'd)

Symbol	Comment
_bootrom.bootkernel.entry	If the emulator hits a hardware breakpoint at the `_bootrom.bootkernel.entry` label, this indicates that device detection or autobaud returned properly.
_bootrom.bootkernel.breakpoint	This is a good address to place a hardware breakpoint. The boot kernel loads a new block header at this breakpoint. The block header can be watched at address 0xFF80 7FF0 or wherever the `pHeader` points to.
_bootrom.bootkernel.initcode	All payload data of the current block is loaded by the time the program passes the `_bootrom.bootkernel.initcode` label. The boot kernel is about to interrogate the `BFLAG_INIT` flag. If set, the initcode can be debugged.
_bootrom.bootkernel.exit	Once the boot kernel arrives at the `_bootrom.bootkernel` exit label, it detects a `BFLAG_FINAL` flag. After some housekeeping, it jumps to the `EVT1` vector.

The boot kernel also generates a circular log file in scratch pad memory. While the `pLogBuffer` and the `dLogByteCount` variables describe the location and dimension of the log buffer, the `pLogCurrent` points to the next free location in the buffer. The log file is updated whenever the kernel passes the `_bootrom.bootkernel.breakpoint` label.

At each pass, nine 32-bit words are written to the log file, as follows.

- block code word (`dBlockCode`) of the block header
- target address (`pTargetAddress`) of the block header
- byte count (`dByteCount`) of the block header
- argument word (`dArgument`) of the block header
- source pointer (`pSource`) of the boot stream
- block count (`dBlockCount`)
- internal copy of the `dBlockCode` word OR'ed with `dFlags`
- content of the `SEQSTAT` register
- 0xFFFF FFFA (-6) constant

The ninth word is overwritten by the next entry set, so that 0xFFFF FFFA always marks the last entry in the log file.

Most of the data structures used by the boot kernel reside on the stack in scratchpad memory. While executing the boot kernel routine (excluding subroutines), the `P5` points to the `ADI_BOOT_DATA` structure. Type "`(ADI_BOOT_DATA*) $P5`" in the IDE's expression view or window to see the structure content.

Boot Management

Blackfin processor hardware platforms may be required to run different software at different times. An example might be a system with at least one application and one in-the-field firmware upgrade utility. Other systems may have multiple applications, one starting then terminating, to be replaced by another application. Conditional booting is called boot management. Some applications may self-manage their booting rules, while others may have a separate application that controls the process, namely a boot manager.

In a master boot mode where the on-chip boot kernel loads the boot stream from memory, the boot manager is a piece of Blackfin software which decides at runtime what application is booted next. This may simply be based on the state of a GPIO input pin interrogated by the boot manager, or it may be the conclusion of complex system behavior.

Slave boot scenarios are different from master boot scenarios. In slave boot modes, the host masters boot management by setting the Blackfin processor to reset and then applying alternate boot data. Optionally, the host could alter the `BMODE` configuration pins, resulting in little impact to the Blackfin processor since the intelligence is provided by the host device.

Booting a Different Application

The boot ROM provides a set of user-callable functions that help to boot a new application (or a fraction of an application). Usually there is no need for the boot manager to deal with the format details of the boot stream.

These functions are:

- BFROM_MEMBOOT discussed in "Flash Boot Modes" on page 17-62 and "SDRAM Boot Mode" on page 17-66
- BFROM_TWIBOOT discussed in "TWI Master Boot Mode" on page 17-77
- BFROM_SPIBOOT discussed in "SPI Master Boot Modes" on page 17-69
- BFROM_OTPBOOT discussed in "OTP Boot Mode" on page 17-85
- BFROM_NANDBOOT discussed in "NAND Flash Boot Mode" on page 17-88

The user application, the boot manager application, or an initcode can call these functions to load the requested boot data. Using the BFLAG_RETURN flag the user can control whether the routine simply returns to the calling function or executes the loaded application immediately.

These ROM functions expect the start address of the requested boot stream as an argument. For BFROM_MEMBOOT, this is a Blackfin memory address, for BFROM_TWIBOOT and BFROM_SPIBOOT it is a serial address. The SPI function can also accept the code for the GPIO pin that controls the device select strobe of the SPI memory.

Multi-DXE Boot Streams

If the start addresses of all the boot streams are predefined, the boot manager needs only to call the ROM functions directly. However since the addresses tend to vary from build to build they may have to be calculated at runtime.

In the world of the elfloader, a boot stream is always generated from a DXE file. It is therefore common to talk about multi-DXE or multi-application booting. When the elfloader utility accepts multiple DXE files on its command line, it generates a contiguous boot image by default. The second boot stream is appended immediately to the first one. Since the utility updates the `ARGUMENT` field of all `BFLAG_FIRST` blocks, the `ARGUMENT` field of a `BFLAG_FIRST` block is called next-DXE pointer (NDP).

The next-DXE pointer of the first DXE boot stream points relatively to the start address of the second DXE boot stream. A multi-DXE boot image can be seen as a linked list of boot streams. The next-DXE pointer of the last DXE boot stream points relatively to the next free address. This is illustrated by an example shown in the next two figures. Figure 17-9 on page 17-57 shows a commented sketch as an example. Figure 17-10 on page 17-58 shows a screenshot of the Blackfin loader file viewer utility for the same example. The `LdrViewer` utility is not part of the CrossCore Embedded Studio or VisualDSP++ tools suite. It is a third-party freeware product available on www.dolomitics.com.

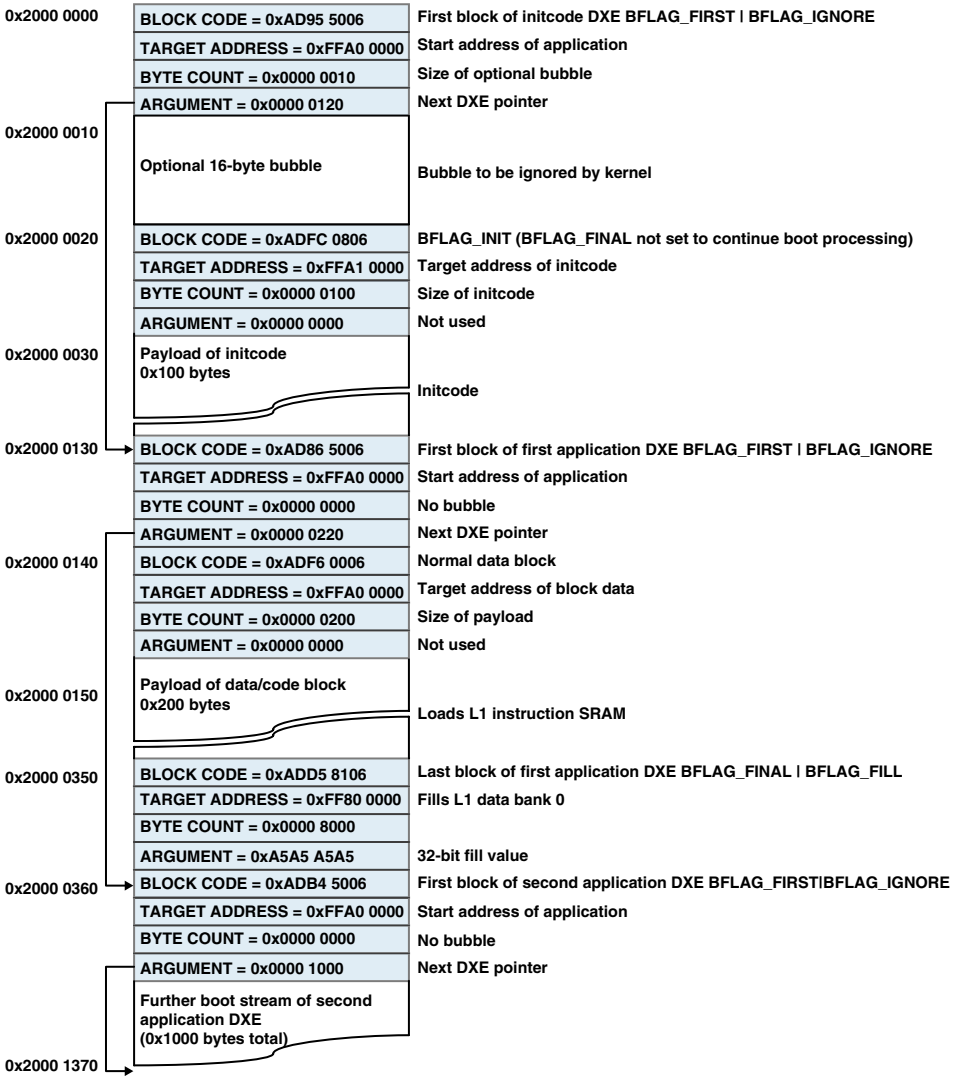


Figure 17-9. Multi-DXE Boot Stream Example for Flash Boot

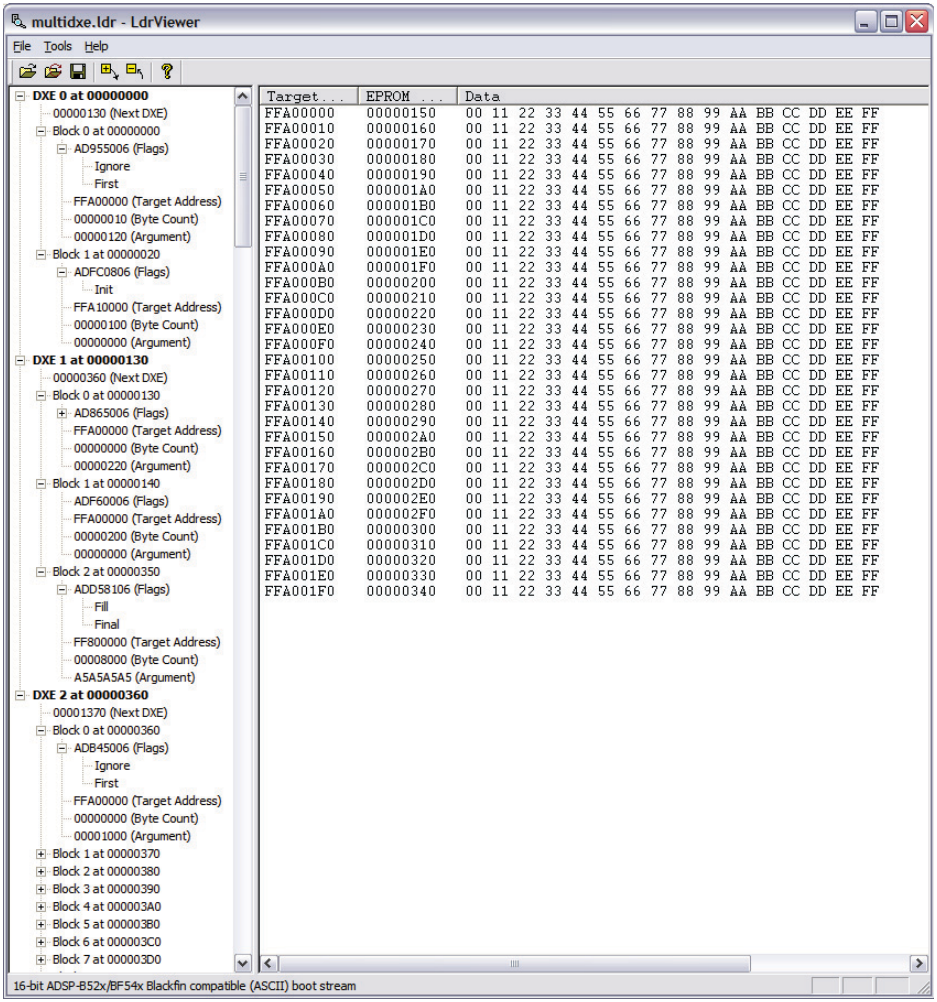


Figure 17-10. LdrViewer Screen Shot

Boot management principles are not only applicable to multi-DXE boot streams. The same scheme, as shown in Figure 17-11 on page 17-59, can be applied to direct code executions of multiple applications. See "Direct Code Execution" on page 17-37 for more information. The example shows a linked list of initial block headers that instruct the boot kernel to

terminate immediately and to start code execution at the address provided by the TARGET ADDRESS field of the individual blocks. There is nothing in the boot ROM that prevents multi-DXE applications from mixing regular boot streams and direct code execution blocks.

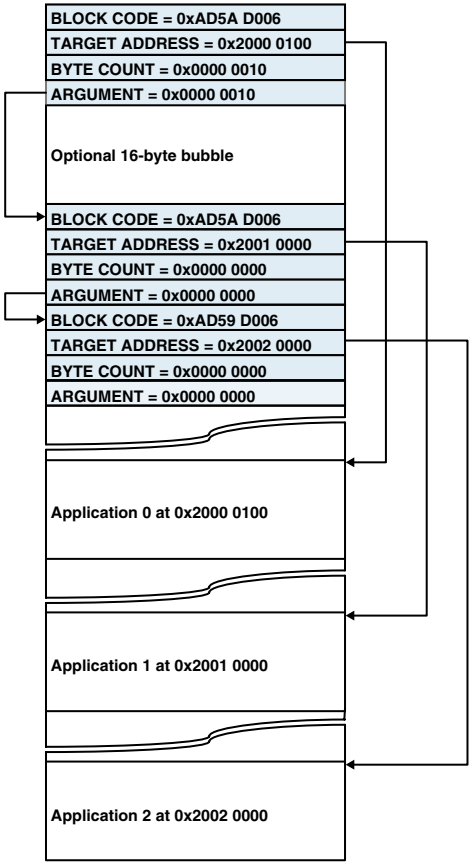


Figure 17-11. Multi-DXE Direct Code Execution Arrangement Example

Determining Boot Stream Start Addresses

The ROM functions BFROM_MEMBOOT, BFROM_TWIBOOT, BFROM_SPIBOOT, etc. not only allow the application to boot a subroutine residing at a given start address, they also assist in walking through linked multi-DXE streams.

When the BFLAG_NEXTDXE bit in dFlags is set and these functions are called, the system does not boot but instead walks though the boot stream following the next-DXE pointers. The dBlockCount parameter can be used to specify the DXE of interest. The routines then return the start address of the requested DXE's boot stream.

Initialization Hook Routine

When the ROM functions BFROM_MEMBOOT, BFROM_SPIBOOT, etc. are called, they create an instance of the ADI_BOOT_DATA structure on the stack and fill the items with default values. If the BFLAG_HOOK is set, the boot kernel invokes a callback routine which was passed as the fourth argument of the ROM routines, after the default values have been filled. The hook routine can be used to overwrite the default values. Every hook routine should fit the prototype:

```
void hook (ADI_BOOT_DATA* pBS);
```

The header files define the ADI_BOOT_HOOK_FUNC type the following way:

```
typedef void ADI_BOOT_HOOK_FUNC (ADI_BOOT_DATA*);
```

The hook function also gives access to the DMA load function used by the respective boot mode, which can be used for general purposes at runtime. For example, in the BFROM_SPIBOOT case, an instance of the load function:

```
ADI_BOOT_LOAD_FUNC *pSpiLoadFunction;
```

can be initialized by equipping the hook function with the instruction:

```
pSpiLoadFunction = pBS->pLoadFunction;
```

Specific Boot Modes

This section discusses individual boot modes and the required hardware connections.

The boot modes differ in terms of the booting source— for example whether data is loaded through the SPI or the parallel interface. Boot modes can also be grouped into slave boot modes and master boot modes.

In slave boot modes, the Blackfin processor functions as a slave to any host device, which is typically another embedded processor, an FPGA device or even a desktop computer. Likely, the Blackfin processor $\overline{\text{RESET}}$ input is controlled by the host device. So, usually the host sets $\overline{\text{RESET}}$ first, then waits until the preboot routine terminates by sensing the HWAIT output, and finally provides the boot data.

If a Blackfin processor, configured to operate in any of the slave boot modes, awakens from hibernate, it cannot boot by its own control. A feedback mechanism has to be implemented at the system level to inform the host device whether the processor is in hibernate state or not. The HWAIT strobe is an important primitive in such systems.

In the master boot modes, the Blackfin processor usually does not need to be synchronized and can load the boot data by itself. Master modes typically read from memory. This can be parallel memory such as flash devices, or serial memory that is read through SPI or TWI interfaces.

Memory boot modes should also be differentiated from peripheral boot modes. Boot modes that load boot streams through memory DMA are referred to as memory boot mode, reading data from regular memory. Peripheral modes load boot data through peripherals such as UART, TWI or SPI. With the exception of the FIFO boot, which is a hybrid, all memory boot modes are master modes. The boot source is typically non-volatile memory, such as a flash or EPROM device or even on-chip ROM. When supported by the system in warm boot scenarios, the boot source can also be SRAM or SDRAM.

Whether from the host (slave booting mode) or from memory (master booting mode), the boot source does not need to know about the structure of the boot stream. However in the case of Host DMA boot, the size (BYTE COUNT) of the boot stream should be known. This is because, having much more control over the Blackfin processor, the host must know what data is to be loaded to specific addresses.

No Boot Mode

When the BMODE pins are all tied low (BMODE = 0000), the Blackfin processor does not boot. Instead it processes factory-programmed OTP pages, then executes an IDLE instruction, preventing it from executing any instructions provided by the regular boot source. The purpose of this mode is to bring the processor up to a clean state after reset.

This mode helps to recover from malicious OTP configuration since it prevents execution of the user-controllable portion of the preboot routine.

When connecting an emulator and starting a debug session, the processor awakens from an idle due to the emulation interrupt and can be debugged in the normal manner.

The no boot mode is not the same as the bypass mode featured by the ADSP-BF53x Blackfin processor. To simulate that bypass mode feature using BMODE = 0001, see "Direct Code Execution" on page 17-37 and "Direct Code Execution" on page 17-155.

Flash Boot Modes

These booting modes are intended to boot from flash or EEPROM memories or even from battery-buffered SRAMs. The flash boot modes are activated by BMODE = 0001. Although this is a single BMODE setting, the ADSP-BF54x processor Blackfin products support various configurations.

- Boot from 8-bit asynchronous flash memory
- Boot from 16-bit asynchronous flash memory
- Boot from 16-bit asynchronous page-mode NOR flash memory
- Boot from 16-bit asynchronous burst-mode NOR flash memory

By default, the boot kernel does not alter any EBIU registers. Therefore, traditional asynchronous flash is assumed and maximum wait states are applied. By programming OTP half pages PBS00L and PBS00H, the user has the option to instruct the preboot routine to alter the EBIU registers as desired. In this way, the EBIU can be preset to access the flash device in either page mode or burst mode. There are also options to customize bus settings, such as wait states and ARDY behavior.

After the preboot routine returns and HWAIT is deasserted the first time, the boot kernel loads an initial burst of four 16-bit words. Then it interrogates the DMACODE field in the byte loaded from the 0x2000 0000 address. For flash mode, the DMA options shown in Table 17-10 are supported.

Table 17-10. DMA Options

DMACODE	DMA Width	Source Modify	Comment
1	8	1	Not recommended Provides ADSP-BF533 style 8-bit boot from 16-bit flash memory
2	8	2	8-bit MDMA boots from 8-bit flash mapped to lower byte of address bus.
6	16	2	16-bit MDMA boots from 16-bit flash
10	32	4	32-bit MDMA boots from 16-bit flash

The DMACODE field is filled by the elfloader utility based on boot mode, -width and -dmawidth settings. See *Loader and Utilities Manual* for details.

After the boot kernel has loaded and interpreted the first four 16-bit words, it continues loading the rest of the first block header and processes the boot stream.

Most of the popular page-mode and burst-mode NOR flash devices default to traditional flash mode and are perfectly designed for altering the operating mode along the way. Theoretically, if the user hesitated to customize boot settings through OTP programming, there was still the option to start booting in traditional asynchronous mode and to alter EBIU settings through an initcode which is loaded and executed early in the boot process.

If the preboot features are not used and the NOR flash device is put into burst or page mode, it must be programmed back to the standard mode before the processor is reset. If the processor can reset itself without software control (through watchdog or double-fault error), a mechanism must be installed that also resets the flash device back to default mode along with the processor. One method to address this is to set the OTP_RESETOUT_HWAIT bit in OTP half page PBS00L and to connect the HWAIT signal to the reset input pin of the NOR flash device.

Hardware configurations for the individual modes are shown in Figure 17-12 and Figure 17-13. The chip select is always controlled by the $\overline{\text{AMS0}}$ strobe. This maps the boot stream to the Blackfin processor's address 0x2000 0000.

See the chapter on System Design for connection of page-mode and burst-mode flash devices.

Some flash devices provide write protection mechanisms, which can be activated during the power-up and reset cycles of the Blackfin processor. In the absence of such mechanisms, a pull-up resistor on the $\overline{\text{AMS0}}$ strobe prevents the chip select from floating when the state of the processor is unknown.

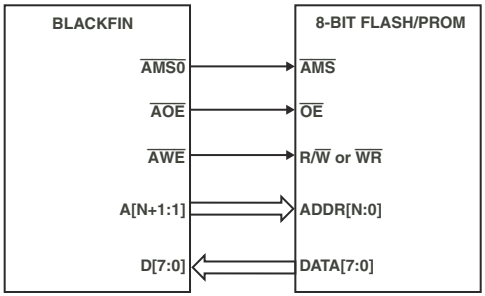


Figure 17-12. 8-Bit Flash Interconnection

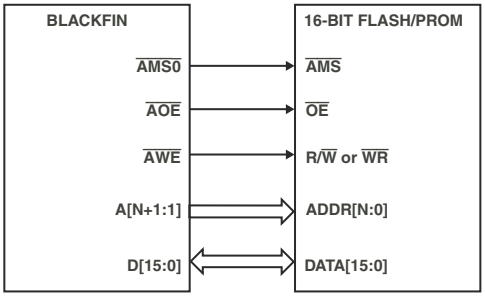


Figure 17-13. 16-Bit Flash Interconnection

In flash mode all the muxed address lines (A4 to A9 on port H and A10 to A25 on port I) are activated by the boot kernel. When BMODE = 0001, none of these pins can function as an input without external hardware protection. Upper address pins are unlikely to toggle and can still be used for GPIO output purposes, with the limitation that the pins are driven low during boot time.

When the EBIU registers are configured to burst-flash mode by the preboot due to OTP programming, the boot kernel activates the NOR clock on the PI15 pin rather than the A25 line.

After RESET has released, the preboot processes a number of OTP pages. Then, the boot kernel starts reading data from the external flash memory. The initial cycles of the flash boot are shown in Figure 17-14. The first

4-word burst loads half of the first boot block header in. After the DMACODE is evaluated the rest of the first block is loaded by the second 4-word burst. As settings are now known the next header is then loaded as an 8-word (16-byte) entity.

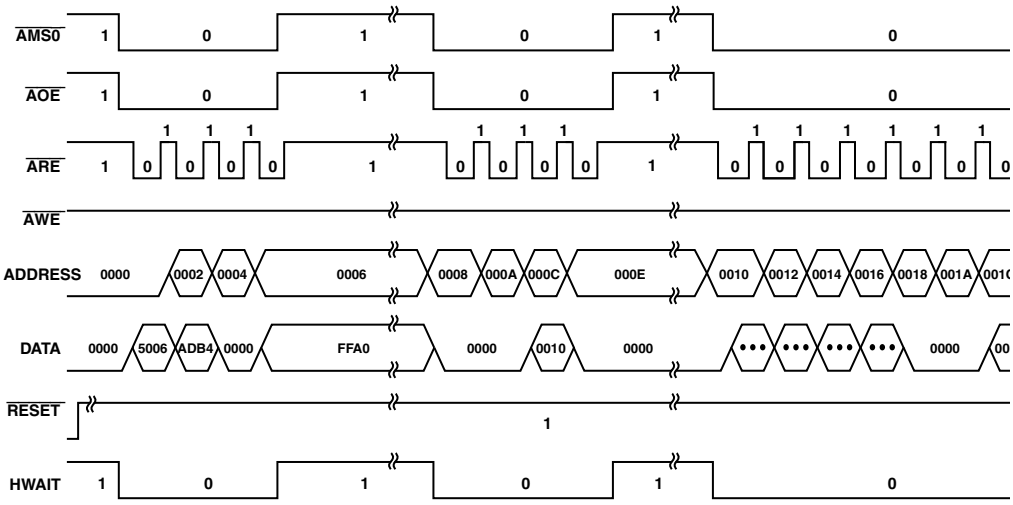


Figure 17-14. 16-bit Flash Mode Waveform

The boot mode BMODE = 0001 can also be used to instruct the boot kernel to terminate immediately and directly execute code from the 16-bit flash memory instead. Code execution from 8-bit flash memory is not supported. See “Direct Code Execution” on page 17-37 for details.

SDRAM Boot Mode

From the boot kernel perspective, the SDRAM boot mode is just another memory boot mode like flash boot. The only differences are that the boot stream is expected at address 0x0000 0010 and the initial eight bytes are loaded by two 32-bit loads.

From the application point of view, SDRAM boot is a completely different scheme. Since SDRAM is volatile memory, BMODE = 1010 is not a valid setting when the processor and the memories have just been powered up. This mode can only be used as a dynamically applied BMODE setting to install warm boot scenarios.

OTP programming is required to boot from SDRAM. Other boot modes can configure the SDRAM controller by execution of an initcode. But in the case of SDRAM boot, the initcode cannot be loaded without having the SDRAM controller already configured.

SDRAM boot is meaningful when the Blackfin processor is in hibernate state or is completely shut off for power savings while the SDRAM is kept alive in self-refresh mode.

Users who prefer to execute code out of SDRAM, rather than performing a boot from it, may refer to "Direct Code Execution" on page 17-37 for details.

FIFO Boot Mode

The FIFO boot mode (BMODE = 0010) boots the Blackfin processor from another processor or FPGA system, referred to as the host device. The host is decoupled from the Blackfin bus by an asynchronous FIFO memory. When compared to the glue-less Host DMA boot modes, the FIFO mode requires less intelligence from the host. The host device is only expected to handshake with the FIFO and to load the entire boot stream in 16-bit portions. There is no need for the host to know about the content and format of the boot stream.

The hardware configuration for the FIFO boot mode is shown in Figure 7-5 on page 7-48. The FIFO chip select connects to the $\overline{\text{AMS3}}$ strobe. Data read requests go to the $\overline{\text{DMAR1}}$ input on pin PH6. The host device controls the Blackfin processor's $\overline{\text{RESET}}$ input. As in all slave modes, the host device should not send requests to $\overline{\text{DMAR1}}$ unless the HWAIT signal

goes inactive. The host device may optionally rely on HWAIT edges to continue or discontinue transmission of boot data in an interrupt controlled manner.

From the boot kernel perspective the FIFO boot mode (BMODE = 0010) is just another memory boot mode, the only exception being that the HMDMA1 block is enabled in advance. Activating this functionality makes the FIFO boot mode become a slave mode.

The bits set in the HMDMA1_CONTROL register are SND, REP and HMDMAEN. The SND bit is new to ADSP-BF54x Blackfin products. The ADSP-BF54x processor's FIFO boot mode differs slightly from the FIFO boot mode provided by the ADSP-BF52x and ADSP-BF53x Blackfin processors.

In the FIFO boot mode, the DMACODE field in the boot block headers must always be 0x06, which instructs the boot kernel to perform 16-bit DMA. The boot kernel increments the applied addresses as if reading from flash memory.

Regardless of the HMDMA settings, the source channel of the memory DMA prefetches four 32-bit words as soon as enabled. Only the transmit channel is stalled and triggered by the HMDMA module. In 16-bit DMA mode, these four early reads translate to eight 16-bit reads.

The ADSP-BF54x processor boot kernel ensures that at least 16 valid data words are ready in the external FIFO—by first counting eight rising edges on the $\overline{\text{DMAR1}}$ request input and then disabling the HMDMA module. When HMDMA is later re-enabled, the prefetch will find valid data and the MDMA can be started safely.

This method requires that the host send 16 more request strobes after it has sent the complete boot stream to the FIFO. This is because the transmit channel of the DMA still has to drain the FIFO, which must be protected from underflow at start.

SPI Master Boot Modes

The SPI boot mode (BMODE = 0011) boots from SPI memories connected to the SPI0SEL1 interface. 8-, 16-, 24-, and 32-bit address words are supported. Standard SPI memories are read using either the standard 0x03 SPI read command or the 0x0B SPI fast read command.

Unlike other Blackfin processors, the ADSP-BF54x processor Blackfin processors have no special support for DataFlash devices from Atmel. Nevertheless, DataFlash devices can be used for booting and are sold as standard 24-bit addressable SPI memories. They also support the fast read mode. If used for booting, DataFlash memory must be programmed in the power-of-2 page mode.

For booting, the SPI memory is connected as shown in Figure 17-15.

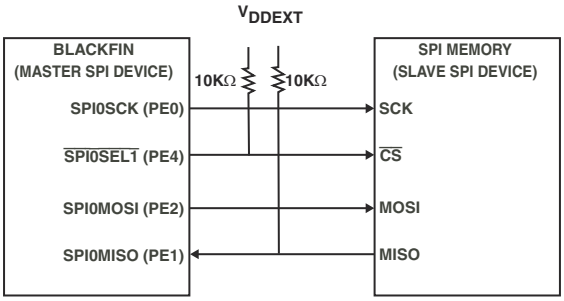


Figure 17-15. Blackfin to SPI Memory Connections

The pull-up resistor on the MISO line is required for automatic device detection. The pull-up resistor on the SPI0SEL1 line ensures that the memory is in a known state when the Blackfin GPIO is in a high-impedance state (for example, during reset). A pull-down resistor on the SPI0SCK line displays cleaner oscilloscope plots during debugging.

For SPI master boot, the SPE, MSTR and SZ bits are set in the SPI0_CTL register. For details see Chapter 22, "SPI-Compatible Port Controllers". With TIMOD = 2, the receive DMA mode is selected. Clearing both the

CPOL and CPHA bits results in SPI mode 0. The boot kernel does not allow SPI0 hardware to control the $\overline{\text{SPI0SEL1}}$ pin. Instead, this pin is toggled in GPIO mode by software. Initialization code is allowed to manipulate the uwSsel variable in the ADI_BOOT_DATA structure to extend the boot mechanism to a second SPI memory connected to another GPIO pin.

By default, the boot kernel sets the SPI0_BAUD register to a value of 133, resulting in a bit rate of SCLK/266. This default value can be altered by programming the 4-bit OTP_SPI_BAUD field in OTP page PBS00L to one of the values in Table 17-11.

Table 17-11. Bit Rate

OTP_SPI_BAUD	SPI_BAUD	Bit Rate
b#0000	133	SCLK/(2x133)
b#0001	Reserved	
b#0010	2	SCLK/(2x2)
b#0011	4	SCLK/(2x4)
b#0100	8	SCLK/(2x8)
b#0101	16	SCLK/(2x16)
b#0110	32	SCLK/(2x32)
b#0111	64	SCLK/(2x64)

Similarly, the boot kernel uses the standard 0x03 SPI read command, by default. Programming the OTP_SPI_FASTREAD bit in OTP page PBS00L enables the fast read mode where the boot kernel uses the 0x0B read command instead and transmits a dummy zero byte after the address bytes.

SPI Device Detection Routine

Since BMODE = 0011 supports booting from various SPI memories, the boot kernel automatically detects what type of memory is connected. To determine whether the SPI memory device requires an 8-, 16-, 24- or 32-bit addressing scheme, the boot kernel performs a device detection sequence prior to booting. The MISO signal requires a pull-up resistor, since the routine relies on the fact that memories do not drive their data outputs unless the right number of address bytes are received.

Initially, the boot kernel transmits a read command (either 0x03 or 0x0B) on the MOSI line, which is immediately followed by two zero bytes. Once the transmission is finished, the boot kernel interrogates the data received on the MISO line. If it does not equal 0xFF (usually a DMACODE value of 0x01 is expected), then an 8-bit addressable device is assumed.

If the received value equals 0xFF, it is assumed that the memory device has not driven its data output yet and that the 0xFF value is due to the pull-up resistor. Thus, another zero byte is transmitted and the received data is tested again. If it differs from 0xFF, either a 16-bit addressable device (standard mode) or an 8-bit addressable device (fast read mode) is assumed.

If the value still equals 0xFF, device detection continues. Device detection aborts immediately if a byte different than 0xFF is received. The boot kernel continues with normal boot operation and it re-issues a read command to read from address 0 again. The first block header is loaded by two read sequences, further block headers and block payload fields are loaded by separate read sequences.

Figure 17-16 illustrates how individual devices would behave.

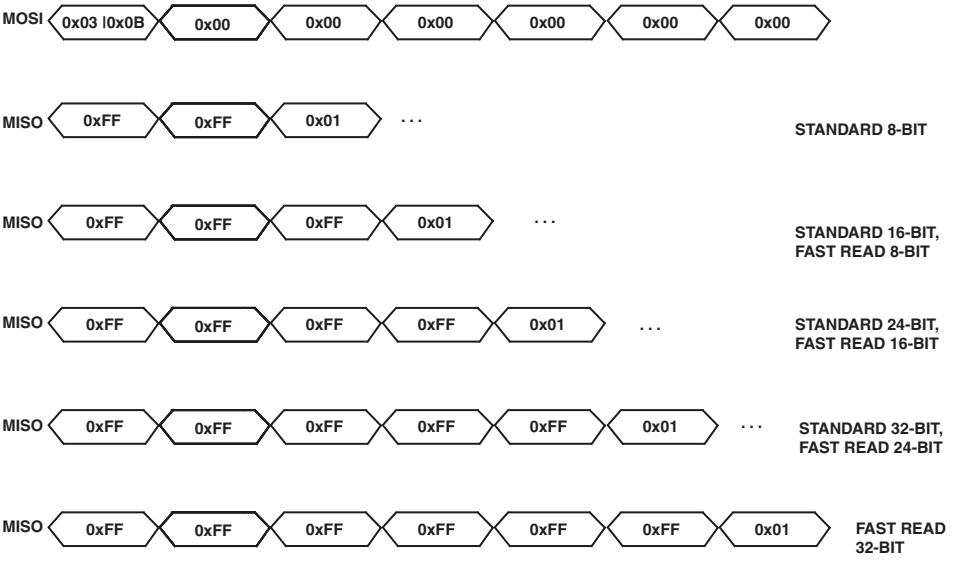


Figure 17-16. SPI Device Detection Principle

Figure 17-17 on page 17-73 shows the initial signaling when a 24-bit addressable SPI memory is connected in SPI master boot mode. After $\overline{\text{RESET}}$ releases and preboot has processed relevant OTP pages, a 0x03 command is transmitted to the MOSI output, followed by a number of 0x00 bytes. The 24-bit addressable memory device returns a first data byte at the fourth zero byte. Then, the device detection has completed and the boot kernel re-issues a 0x00 address to load the boot stream.

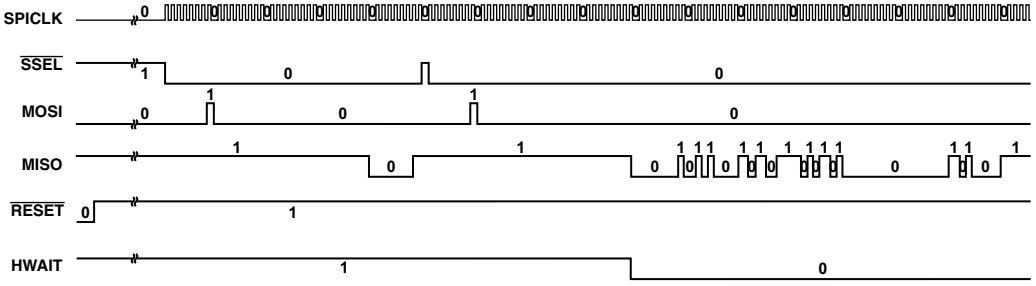


Figure 17-17. Typical SPI Master Boot Waveforms

SPI Slave Boot Mode

For SPI slave mode boot (BMODE = 0100), the Blackfin processor is consuming boot data from an external SPI host device. SPI0 is configured as an SPI slave device. The hardware configuration is shown in Figure 17-18. As in all slave boot modes, the host device controls the Blackfin processor RESET input.

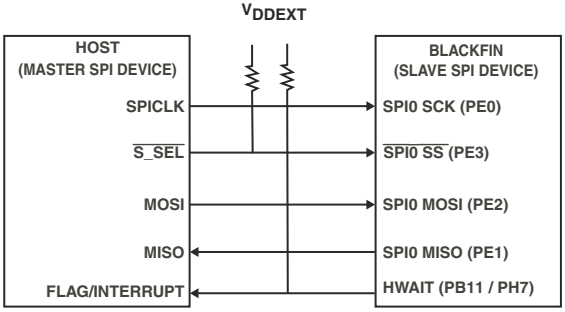


Figure 17-18. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

The host drives the SPI clock and is responsible for the timing. The host must provide an active-low chip select signal that connects to the SPIOSS input of the Blackfin processor. It can toggle with each byte transferred or remain low during the entire procedure. 8-bit data is expected. The 16-bit mode is not supported.

In SPI slave boot mode, the boot kernel sets the CPHA bit and clears the CPOL bit in the SPIO_CTL register. Therefore the MOSI pin is latched on the falling edge of the SPI_SCK pin. For details see Chapter 22, "SPI-Compatible Port Controllers".

In SPI slave boot mode, HWAIT functionality is critical. The HWAIT handshake signal can operate on either the GPIO pin PB11 or on PH7 when the OTP_ALTERNATE_HWAIT in OTP page PBS00L is programmed. When high, the resistor shown in Figure 17-18 programs HWAIT to hold off the host. HWAIT holds the host off while the Blackfin processor is in reset or executing the preboot. Once HWAIT turns inactive, the host can send boot data. The SPI module does not provide very large receive FIFOs, so the host must test the HWAIT signal for every byte. Figure 17-20 on page 17-76 illustrates the required program flow on the host side.

Figure 17-19 on page 17-75 shows the initial waveform for an SPI slave boot case. As soon as the Blackfin processor releases HWAIT after reset, the host device pulls the SPIOSS pin low and starts transmitting data. After the eighth data word has been received, the boot kernel asserts HWAIT again as it has to process the DMACODE field of the first block header. When the host detects the asserted HWAIT it gracefully finishes the transmission of the on-going word. Then, it pauses transmission until HWAIT releases again.

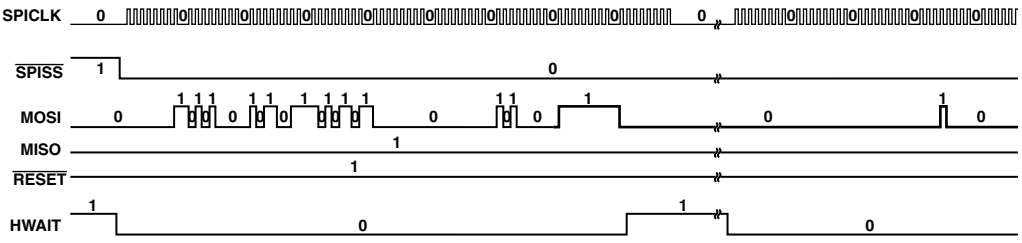


Figure 17-19. Typical SPI Slave Boot Waveforms

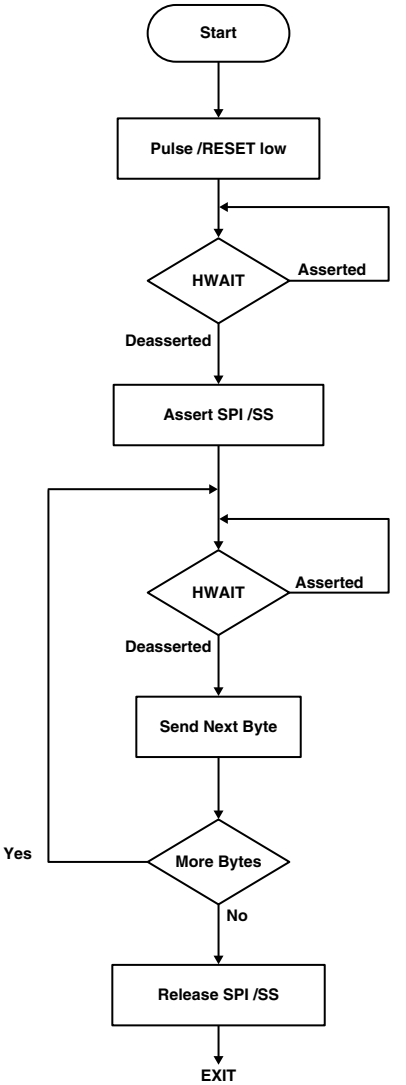


Figure 17-20. SPI Program Flow on Host Device

TWI Master Boot Mode

In TWI master boot mode (BMODE = 0101) the boot kernel reads boot data from I²C memory connected to the TWI0 interface. The Blackfin processor selects the slave EEPROM with the unique ID 0xA0, submits successive read commands to the device starting at internal address 0x0000, and begins clocking data to the processor. The EEPROM's device select bits A2–A0 must be 0s (tied low) when present. The I²C EPROM device should comply with Philips I²C Bus Specification version 2.1 and should have the capability to auto increment its internal address counter such that the contents of the memory device can be read sequentially. Connections are shown in Figure 17-21 "TWI Master Boot Mode Connections" on page 17-77.

On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the BFLAG_INDIRECT bit set. Initcodes can alter the placement of the temporary buffer by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure.

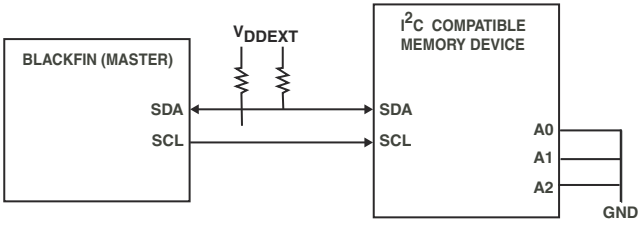


Figure 17-21. TWI Master Boot Mode Connections

The Blackfin processor's TWI controller outputs the address of the I^2C device to boot from, in this case 0xA0, where the least significant bit indicates the direction of the transfer. In this example, it is a write (0) to write the first two bytes of the internal address from which to start booting (0x00).

Figure 17-23 "TWI Init and Fill Block Timing" on page 17-78 shows the TWI init and zero fill blocks.

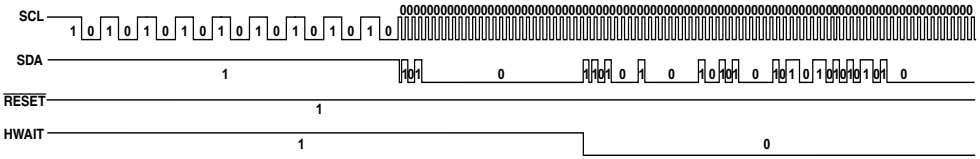


Figure 17-22. TWI Master Boot Timing

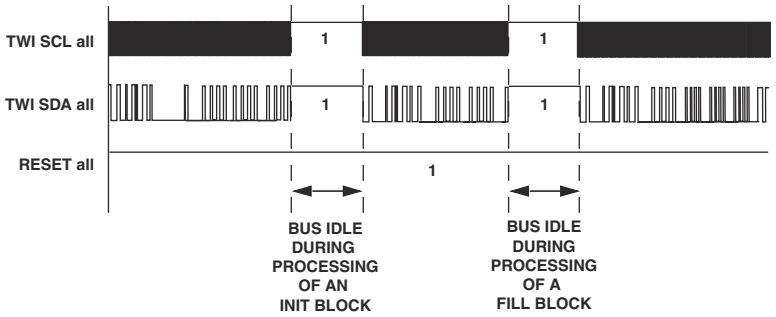


Figure 17-23. TWI Init and Fill Block Timing

Figure 17-22 "TWI Master Boot Timing" on page 17-78 shows the initial waveforms for TWI master boot. After reset, the kernel generates nine slow pulses on the SCL output to ensure the TWI memory's state machine exits any pending state. Then a start condition is issued and 0xA0 address command is issued, where the least significant bit indicates the direction of the write. In this case it is a write (0) in order to write two more 0x00 address bytes.

By default, it is assumed that the I^2C memory device is two-byte addressable. This can be changed by programming the OTP_TWI_TYPE bit field in OTP page PBS00L as shown in Table 17-12 and Figure 17-45 on page 17-111.

Table 17-12. Addressable Bytes

OTP_TWI_TYPE	Address Bytes
00	2
01	3
10	4
11	1

The TWI0 controller is programmed to generate a 30% duty cycle clock in accordance with the I^2C clock specification for fast-mode operation (PRESCALE = 0xA, CLKDIV = 0x811) as shown in Table 17-13. The default values can be altered by OTP programming. Setting the OTP_TWI0_CLKDIV bit in OTP page PBS00L changes the OTP_TWI0_CLKDIV register value to 0x3232 as recommended for 100 kHz TWI operation. The OTP_TWI_PRESCALE field controls the prescale value written to the TWI0_CONTROL register.

TWI Slave Boot Mode

In TWI slave boot mode (BMODE = 0110) the Blackfin processor consumes data from a I^2C host device connected to the TWI0 interface. The I^2C host selects the slave (Blackfin processor) with the 7-bit slave address 0x5F. When the Blackfin processor acknowledges, the host can download the boot stream. The I^2C host should comply with Philips I^2C Bus Specification version 2.1. The host supplies the serial clock.

Table 17-13. Prescale Value

OTP_TWI_PRESCALE	PRESCALE	Recommended[1]
000	0x0A	SCLK = 100 MHz
001	0x0E	SCLK = 140 MHz (theoretical)
010	0x0C	SCLK = 120 MHz
011	0x0A	SCLK = 100 MHz
100	0x08	SCLK = 80 MHz
101	0x06	SCLK = 60 MHz
110	0x04	SCLK = 40 MHz
111	0x02	SCLK = 20 MHz

1 Check the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the maximum SCLK frequency.

Connections are shown in Figure 17-24 “TWI Slave Boot Mode Connections” on page 17-80.

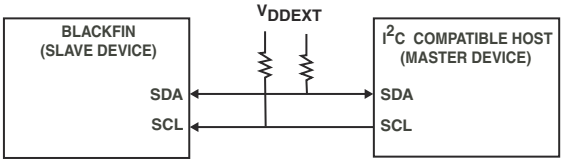


Figure 17-24. TWI Slave Boot Mode Connections

Figure 17-25 “TWI Slave Boot Timing” on page 17-81 shows initial waveforms for TWI slave boot. As soon as HWAIT releases after reset the host starts transmitting the boot stream data. It starts with a start condition and a 0xBE command, which is a composite of the 0x5F address and a trailing zero bit to indicate write direction.

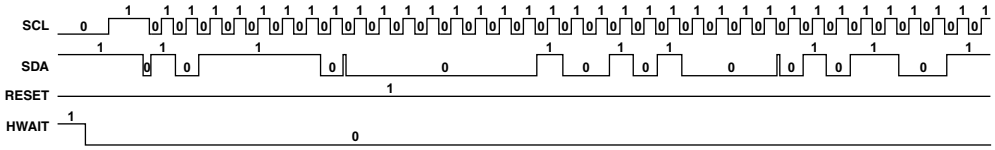


Figure 17-25. TWI Slave Boot Timing

Figure 17-26 on page 17-81 shows an example of bit stretching.

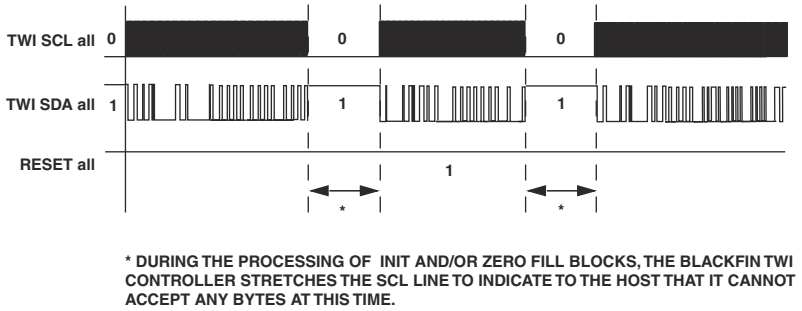


Figure 17-26. TWI Bit Stretching Timing

On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the BFLAG_INDIRECT bit set. Initcodes

can alter the placement of the temporary buffer by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure.

UART Slave Mode Boot

Figure 17-27 on page 17-82 shows the interconnection required for booting. The figure does not show physical line drivers and level shifters that are typically required to meet the individual UART-compatible standards.

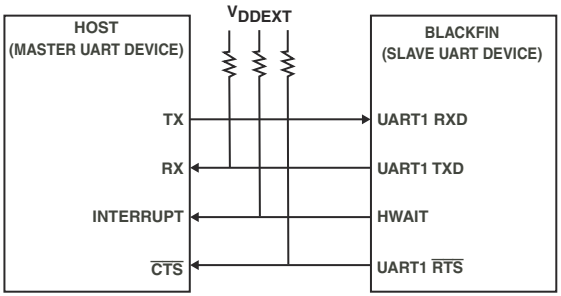


Figure 17-27. UART Slave Boot Mode Connections

For BMODE = 0111, the ADSP-BF54x processor consumes boot data from a UART host device connected to the UART1 interface. Automatic control of the RTS output provides flow control.

The host downloads programs formatted as boot streams using an autobaud detection sequence. The host selects a bit rate within the UART clocking capabilities. To determine the bit rate when performing the autobaud, the boot kernel expects an "@" character (0x40, eight data bits, one start bit, one stop bit, no parity bit) on the UART RXD input. The boot kernel acknowledges, and the host then downloads the boot stream. The acknowledgement consists of four bytes: 0xBF, UARTx_DLL, UARTx_DLH, 0x00. The host is requested to not send further bytes until it has received the complete acknowledge string. Once the 0x00 byte is received, the host can send the entire boot stream. The host should know the total byte

count of the boot stream, but it is not required to have any knowledge about the content of the boot stream. Further information regarding autobaud detection is given in "Autobaud Mode" on page 10-33.

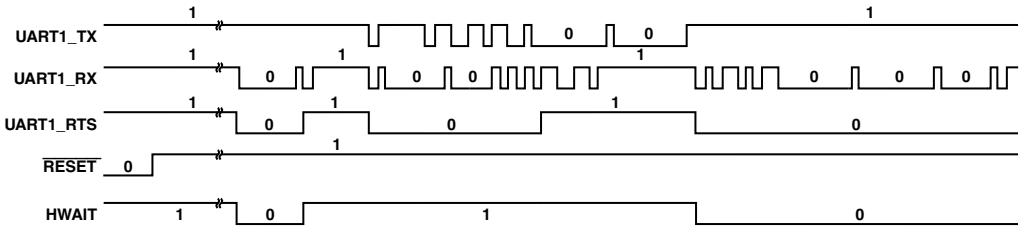


Figure 17-28. UART Autobaud Waveform

When the boot kernel is processing fill or initcode blocks it might require extra processing time and needs to hold the host off from sending more data. This is signalled with the HWAIT output as well as by the RTS output. When equipped with a pull-up resistor the HWAIT signal imitates the behavior of an RTS output and could be connected to the CTS input of the booting host. The host is not allowed to send data until HWAIT turns inactive after a reset cycle. Therefore a pulling resistor on the HWAIT signal is required.

If the resistor pulls to ground, the host must pause transmission when HWAIT is low and is permitted to send when HWAIT is high. A pull-up resistor inverts the signal polarity of HWAIT. The host should test HWAIT at every transmitted byte.

During ADSP-BF54x boot operation, the host device more likely relies on the RTS output of UART1. Then, the use of HWAIT becomes optional. At boot time the Blackfin does not evaluate RTS signals driven by the host and the UART1 CTS input is inactive. Since the RTS is in a high impedance state when the Blackfin processor is in reset or while executing preboot, an external pull-up resistor to VDDEXT is recommended.

Figure 17-29 on page 17-84 plus Figure 17-30 on page 17-85 show the initial case of the UART boot mode. As soon as HWAIT releases after reset, the boot kernel expects to receive a 0x40 byte for bit rate detection. After the bit rate is known, the UART is enabled and the kernel transmits for bytes.

Figure 17-29 on page 17-84 and Figure 17-30 on page 17-85 compare $\overline{\text{RTS}}$ and HWAIT timing when an extended initcode executes. Since code execution distracts from data loading, the host device should be prevented from sending more data. The HWAIT timing is much more conservative than the $\overline{\text{RTS}}$. If the host relies on HWAIT, the UART receive buffer may not be filled over watermark level and $\overline{\text{RTS}}$ might not be de-asserted at all. If, however, the host relies on $\overline{\text{RTS}}$ it will be stalled a couple of bytes later. Both methods are valid.

As shown in Figure 17-30, when the UART is enabled, $\overline{\text{RTS}}$ goes low, encouraging the host to send the boot stream data immediately. With a half-duplex UART connection this must be avoided. The host should either rely on the HWAIT signal or wait until it has received the four bytes from the Blackfin processor, before sending any data.

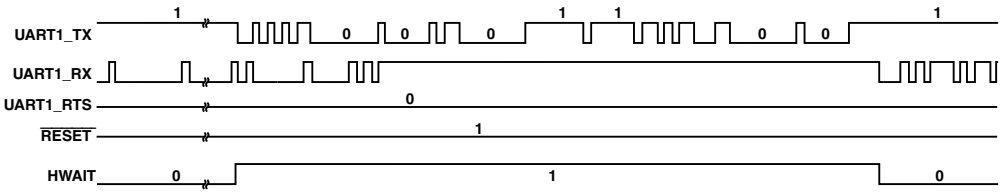


Figure 17-29. UART Boot - Host relying on HWAIT

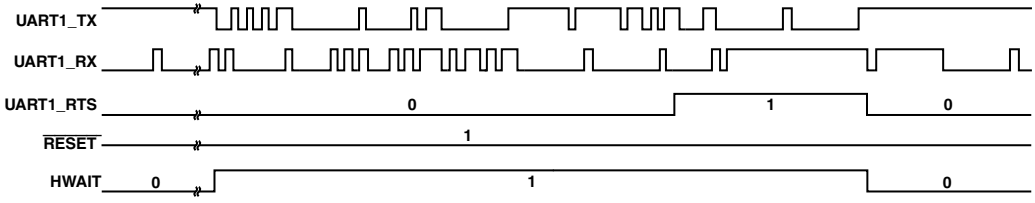


Figure 17-30. UART Boot - Host relying on RTS

For UART boot, it is not obvious on how to change the PLL by an initcode routine. This is because the UARTx_DLL and UARTx_DLH registers have to be updated to keep the required bit rate constant after the SCLK frequency has changed. It must be ensured that the host does not send data while the PLL is changing. The initcode examples provided along with the CCES or VisualDSP++ tools installation demonstrate how this can be accomplished.

OTP Boot Mode

In the OTP boot mode (BMODE = 1011), the boot kernel loads the boot stream from the on-chip OTP memory. OTP booting is a self-sufficient booting mechanism that does not require external boot memory or a host device.

By default the boot kernel starts loading the boot stream starting from OTP page 0x40. This is in the public OTP region. The boot stream can occupy all pages up to OTP page 0xDF, resulting in a boot stream length of up to 2560 bytes. The start address of the boot stream can be altered by programming the OTP_START_PAGE field in the PBS01H page. If there is no conflict with the alternate preboot pages feature, the OTP_START_PAGE field can be set to 0x20, resulting in a boot stream length of up to 3072 bytes.

In the current implementation, the OTP engine has no DMA support. Data is loaded and copied by core instructions. Nevertheless the DMACODE field should be set to 0xA, indicating 32-bit operation. The boot kernel ensures proper operation at 32-bit granularity, but 64-bit alignment may help to reduce the number of OTP pages that have to be read during boot processing. Byte 0 of the boot stream is expected to be byte 0 of the lower 32-bit word of the lower 0x40 half page.

In the OTP boot mode, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the BFLAG_INDIRECT bit set. Initcodes can alter the placement of the temporary buffer by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure.

Host DMA Boot Modes

The Host DMA boot modes differ completely from other boot modes because the boot kernel has no control over the DMA channels. The host device masters the DMA, so the host device must parse the boot stream by itself.

The two host DMA boot modes (BMODE = 1110 for 16-bit and BMODE = 1111 for 8-bit) are almost identical. The differences are the port muxing control and the initial programming of the HOST_CONTROL register. The 16-bit boot mode uses the HOSTDP's acknowledge mode while the 8-bit boot mode sets the INT_MODE bit in the HOST_CONTROL register to activate the interrupt mode.

Connection of a host device to the Blackfin processor is discussed in Chapter 8, "Host DMA Port". For booting, the host device should control the $\overline{\text{RESET}}$ of the Blackfin processor. The host processor must poll the

HOST_STATUS register using a configuration read of the HOSTDP until the ALLOW_CNFG bit is set (indicating that the host may begin sending the 7 configuration words). This is necessary before each configuration of the HOSTDP. The host processor may optionally sense the HWAIT signal to determine when it should begin polling the ALLOW_CNFG bit.

The HOSTDP interface does not support the advanced boot kernel operations such as fill, CRC or callback. There is simple support to simulate the initcode functionality. Typically, this feature is not so important when the preboot OTP memory pages can be programmed to configure the PLL and SDRAM controllers. However, if the user does not have the option to program OTP memory, the simulated initcode is the only option to speed up the processor clocks and to enable the SDRAM controller for booting. One of these options must be used for the host device to boot into SDRAM memory.

In order to simulate initcodes the host device must send a valid initcode routine to L1 instruction address 0xFFA0 0000. Additionally, the host is required to issue an HIRQ command after sending the 7 configuration words (but before sending any data) for the initcode block to the HOSTDP. Once the boot kernel detects an HIRQ command from the host and the DMA work unit is complete, the boot kernel will issue a CALL instruction to the address held in the EVT1 register, and the C language initcode routine is called. EVT1 defaults to 0xFFA0 0000, but it can be modified by user instructions during the boot process. When the initcode returns, the regular boot process continues. This can be repeated multiple times if necessary.

If the initcode routine has properly configured the SDRAM controller, subsequent Host DMA work units can write to SDRAM memory. Similarly, if the initcode has programmed the PLL, the Host DMA port can run at higher speed since it is SCLK dependent.

The same scheme is used to terminate the boot process. When the host is ready to send the final boot block of the application it needs to send the 7 configuration words required by the HOSTDP. The host device should

then send an HIRQ command followed by the remaining data. Once all data has written, the boot kernel executes another CALL instruction and the application takes control of the system rather than returning to the boot kernel.

Figure 17-31 on page 17-89 illustrates boot kernel processing in the Host DMA boot mode. Figure 17-32 on page 17-90 illustrates host device flow.

NAND Flash Boot Mode

NAND flash boot mode (BMODE = 1101) is intended to boot from SLC NAND flash memory devices connected directly to the NAND Flash Controller (NFC) of the ADSP-BF54x processor processors.

By default the NAND flash boot mode configures the read and write delay strobe timing parameters within the NFC_CTL register with RD_DLY = 0x3 and WR_DLY = 0x3. This provides t_{RP} and t_{WP} timings of four SCLK cycles (30ns at 133 MHz) to provide maximum compatibility. By programming OTP half page PBS01H, the user has the option to instruct the preboot routine to provide alternate settings prior to accessing the NAND flash for the first access. In NAND flash boot mode, the HWAIT signal does not toggle. The respective GPIO pins remain in high-impedance mode.

Providing OTP configurations of RD_DLY = 0x0 and WR_DLY = 0x0 will result in the boot kernel using the default configuration of RD_DLY = 0x3 and WR_DLY = 0x3. The highest performance settings for NAND flash boot are enabled with WR_DLY = 0x1 and RD_DLY = 0x0.

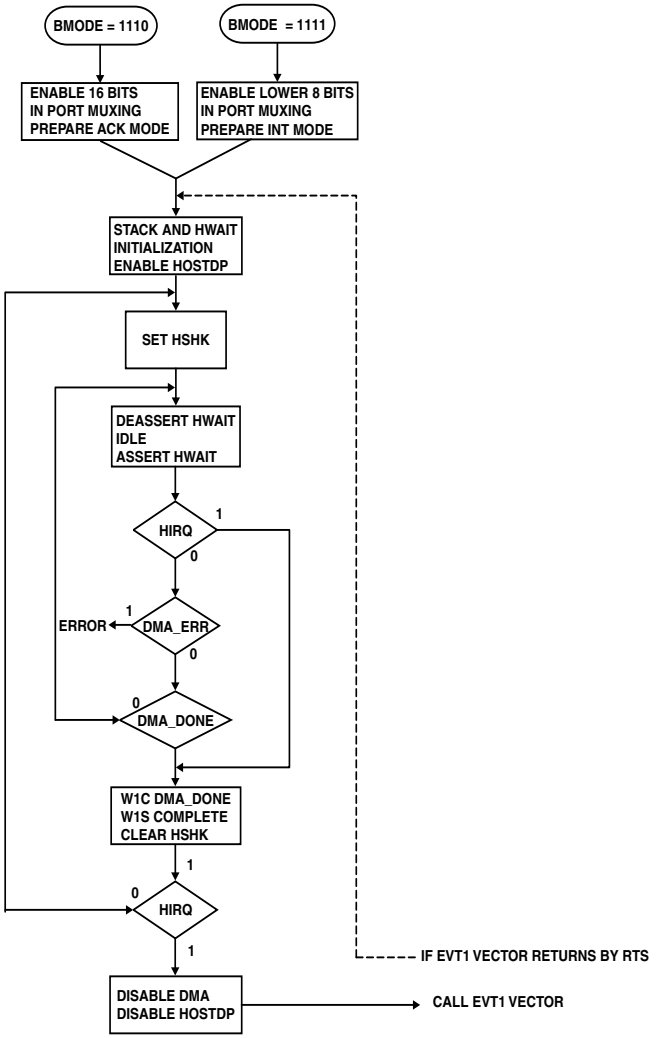


Figure 17-31. Boot Kernel Processing in Host DMA Boot Mode

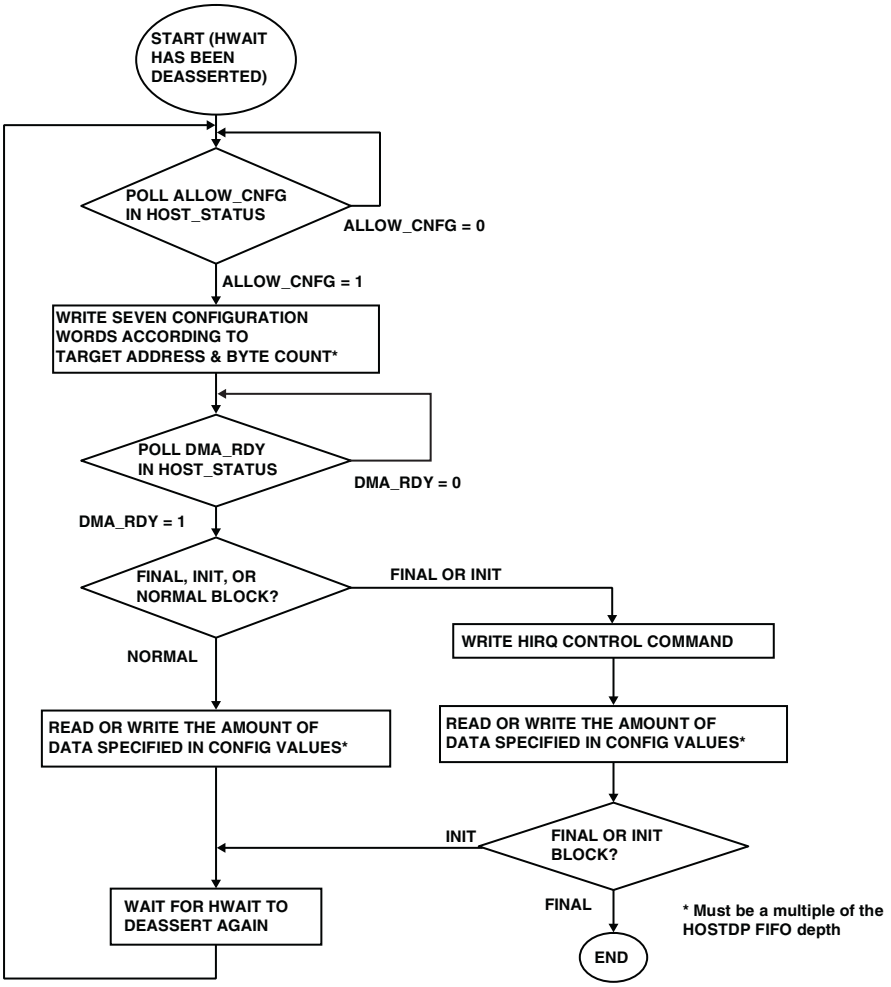


Figure 17-32. Host Device Flow in Host DMA Boot Mode

Supported Devices

NAND flash boot provides support for booting from a number of NAND flash devices from a number of different manufacturers. There are two main classifications of single SLC NAND flash memories:

- small-page NAND flash
- large-page NAND flash (8-bit and 16-bit)

The small-page NAND flash devices use a different addressing scheme for accessing the NAND flash array than that required by large-page NAND flash devices. Additionally small-page devices require a different command set for reading from different parts of the page.

For booting, small-page devices must comply with the array configuration in Table 17-14 and support the commands in Table 17-15.

Table 17-14. Supported Small-page Device Array Configuration

Parameter	Size
Page Size	512 Bytes
Block Size (excluding spare area)	16384 Bytes (32 pages)
Spare Area	16 Bytes
1st half of page	256 Bytes
2nd half of page	256 Bytes
Maximum number of addressable blocks	524288

Table 17-15. Supported Small-page Commands

Operation	Command
Reset	0xFF
Read from 1st half of array	0x00
Read from 2nd half of array	0x01
Read from spare area	0x50

The NAND flash boot kernel, by default, issues four address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure in an initialization routine that is executed before the main application boot stream is processed. To load the initialization function however, four address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.

NAND flash boot provides support for a number of large-page array configurations. The fourth byte of the NAND flash Electronic Signature is used to configure the boot kernel for correct access to the memory array. The boot kernel supports any large-page NAND flash device whose Electronic Signature fourth byte is complies with the format in Figure 17-33.

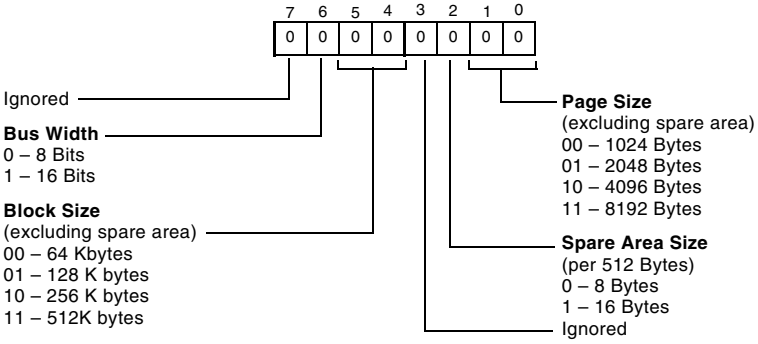


Figure 17-33. Supported 4th byte of NAND Flash Electronic Signature

Table 17-16 shows the large-page command set that is supported.

Table 17-16. Supported Large-page Commands

Operation	1st Command	2nd Command
Reset	0xFF	
Read Electronic Signature	0x90	
Read	0x00	0x30

Due to the auto detection method used, large-page NAND flash devices must not react to the issuing of command 0x50 followed by four address cycles by driving the R $\overline{B}$ signal low and then high again.

The NAND flash boot kernel, by default, issues five address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure in an initialization routine that is executed before the main application boot stream is processed. To load the initialization function however, five address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.

Supported 16-Bit NAND flash memories must only use the lower eight bits of the bus for the command and address cycles. 16-bit command and address cycles are not supported.

Hardware configuration for the NAND flash boot mode is shown in Figure 17-34.

As the GPIO pins were originally configured as inputs, a pull-up resistor is required on PJ1\$\overline{\text{ND_CE}}$. This ensures that the device is not selected after a reset before the required GPIO pins have been configured correctly.

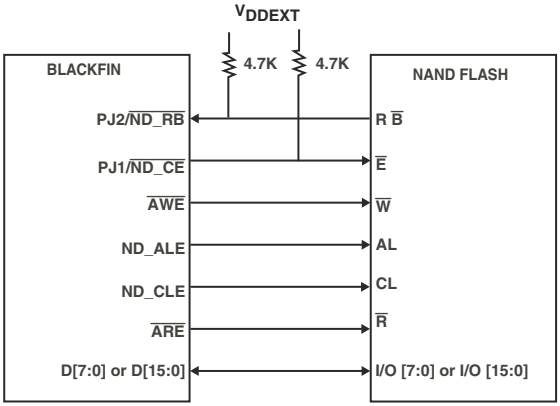


Figure 17-34. 8-Bit/16-Bit NAND Flash Interconnection

NAND Flash Page Structure

The NAND flash boot option transfers data contents from the main area of the NAND flash using a 256 byte DMA transfer. The spare area section at the end of the page contains the ECC error checking parity data for each 256 sub block of the main data area. The spare area section is divided into equal sizes corresponding to the number of 256 byte blocks contained within a page. For a 512 Mbyte small-page device, the spare area is divided into two sections.

The first three bytes of each sub section of the spare area contains the 22-bit ECC parity data for the corresponding data block. The very last byte of the spare area is reserved in the first and second pages of each block for the bad-block marker. Figure 17-35 shows the page structure for a NAND flash device with a page size of 2048-bytes. The 64-byte area is divided into eight 8-byte sections. The first three bytes of each section contains the parity data for the corresponding 256-byte block.

The last byte in the page is used as the bad-block marker in the event that the device only contains 8-bytes of spare area per 512-byte block instead of the more common 16-bytes per 512-byte block.

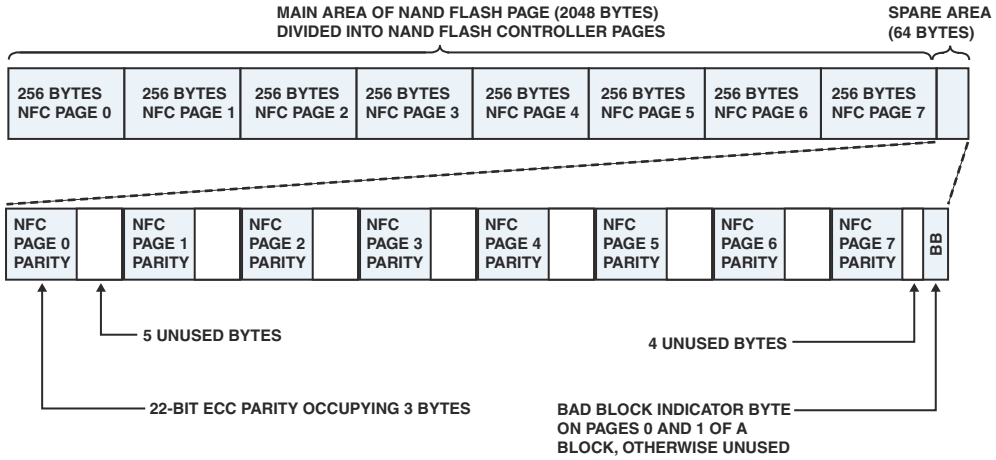


Figure 17-35. Page Layout of NAND Flash Device, 2048-Byte Page Size

Auto Detection

Once the boot kernel has detected the NAND flash boot option, the first operation to be performed is the auto detection procedure of the NAND flash device.

The boot kernel first issues a reset signal to the NAND flash device. The reset command brings the NAND flash out of the default read mode—ready to accept a command. The NAND flash reacts by driving the R $\overline{B}$ signal low and then high again.

The processor, after issuing the reset command, enters a nested loop that checks the status of the R $\overline{B}$ signal every 100 SCLK cycles. A maximum of 100 checks are performed. If the ready busy signal is not driven low and then high again after 100 attempts, then the boot kernel enters the safe idle mode as it assumed that no NAND flash device is present. The loop terminates when R $\overline{B}$ assertion is detected. The processor then proceeds to determine if the attached device is a small-page NAND flash device.

Small-page device detection consists of issuing a command to read from the spare area of the device, command 0x50, followed by four address cycles. Once again the processor enters a nested loop routine waiting for detection of a rising edge of the $\overline{\text{ND\_RB}}$ signal. If the rising edge is detected then the boot kernel is configured to boot from the supported small-page device.

If no rising edge is detected by the time the loop terminates, the device is assumed to be a large-page device. The processor issues another reset command to reset the large-page device, then proceeds to read the Electronic Signature to configure the boot kernel appropriately.

Boot Stream Processing

To successfully boot from NAND flash, blocks of data must be first transferred to the processors internal memory to be processed by the boot kernel. A 512 byte temporary storage space located at 0xFF907E00 – 0xFF907FFF is used.

This storage space is split into two buffers each consisting of 256 bytes.

The 256 byte buffer at location 0xFF907E00 – 0xFF907EFF is referred to as the “MainBuffer”. The remaining 256 bytes from 0xFF907F00 – 0xFF907FFF are referred to as the “PrefetchBuffer”.

As the 0xFF907E00 – 0xFF907FFF address range is usable Data Bank B memory, the application itself must not resolve anything to this space. Take care to omit this region from the defined memory range in the application’s LDF file to prevent boot-time conflicts from overwriting these buffers.

The NAND flash controller is configured for a 256 byte page size. During the boot phase, a single block transfer consists of 256 bytes. All block transfers from the NAND flash device go the PrefetchBuffer. The boot kernel determines if the MainBuffer is empty, is partially processed or is fully processed. If the MainBuffer is empty or all data currently residing in

the MainBuffer is processed, the boot kernel copies the contents of the PrefetchBuffer into the MainBuffer—then requests another 256 block of data from the NAND flash. This process continues until the entire boot stream is processed.

An important requirement of NAND flash devices is the need for error checking and correction (ECC) on the received data. The NAND flash controller of the ADSP-BF54x processor devices uses a Hamming Code algorithm to automatically generate two sets of parity data for each 256 byte block transfer. The two sets of parity data each consist of 11 bits providing a total of 22 bits of parity data. This allows for the detection and correction of a single bit error within a 256 byte block, detection of a double error, and detection of an error within the parity data itself.

The boot kernel uses the embedded NFC ECC parity generation hardware and performs the error correction algorithm after every block transfer to the PrefetchBuffer. The kernel detects when the requested data resides in a new page. Before requesting the actual data, the kernel reads the data from the spare area section of the page, where the ECC parity data resides, to the PrefetchBuffer. Then the kernel stores the data internally on the stack to the EccParity structure.

The parity data for the entire NAND flash page is stored, allowing for error checking to be performed on all further data transfers from that page without requiring further access to the spare area. Thus the kernel adopts a more efficient access method when requesting the actual data, by only issuing a single read command for sequential 256 byte block accesses to a page.

Because the NAND flash boot procedure uses a prefetch mechanism, the 256 byte block following the end of the boot stream must have the correct ECC parity field programmed. Failure to adhere to this results in the boot kernel generating an uncorrectable error when fetching the block of data; and the boot process terminates.

Software Configurable NAND Flash Boot Modes

The NAND flash boot mode supports three different boot methods with regards to handling errors and bad blocks.

- Sequential Block Mode (default)
- Block Skip Mode
- Multiple Image Mode

The three booting options provide users with flexibility in how they use a NAND flash for booting purposes.

The three boot modes are configured through the `uwBlockSkipFeature` variable of the `EccParity` structure. By default `uwBlockSkipFeature` = 0, configuring the device for Sequential Block Mode. The user can change the boot mode by modifying the `uwBlockSkipFeature` variable in an initialization routine that is loaded and executed before the main application boot stream is processed. Access to the `ADI_BOOT_NAND` structure is provided by a pointer stored in the `dUserLong` parameter of the `ADI_BOOT_DATA` structure.

Sequential Block Mode

The default boot method is the Sequential Block Mode. In this mode no bad block detection is performed. The processor simply boots the boot stream starting from page 0 of block 0 until the end of the boot stream is reached. Error correction is always performed for greater reliability. However, if an uncorrectable error or error in the parity data is detected, the booting process terminates and the error handler is called.

This boot mode is suited for applications that wish to adopt a second stage boot loader approach, where the second stage loader starts from the first byte in the NAND flash.

If the boot stream to be loaded spans a number of blocks then all blocks that the boot stream occupies must be good blocks. If a block is known to be bad then this boot method should not be adopted for that particular device.

Figure 17-36 shows some typical usage scenarios for this mode.

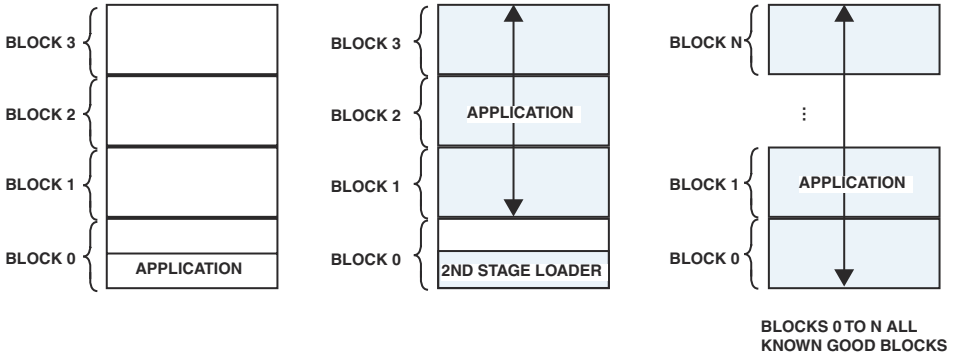


Figure 17-36. Sequential Block Mode Usage Scenarios

Block Skip Mode

This mode is enabled with `uwBlockSkipFeature` = 1. When enabling this mode the user must also set `uwBlockModifier` = 1. Failure to do so can result in the boot procedure failing.

This boot mode is suited for larger applications not adopting the second stage loader approach. During the loading of the application to the NAND flash, upon detection of factory set bad block, the last byte of the spare area of the first and second page of the bad block is set to a non 0xFF value.

The boot procedure works in a similar manner to the Sequential Block Mode except on detection of an access to a new block the spare area sections of the first two pages are loaded. The boot kernel checks the last byte of each. If either is not equal to 0xFF then the page is detected as bad. A

byte offset of 1 block is then applied to all subsequent data requests thus skipping any bad blocks. This allows for the booting a single larger stream that is impeded by bad blocks in the area that the boot stream occupies.

Each time a bad block is encountered the byte offset applied to the address of the requested data is incremented by 1 block. Figure 17-37 highlights a typical usage scenario for this boot method.

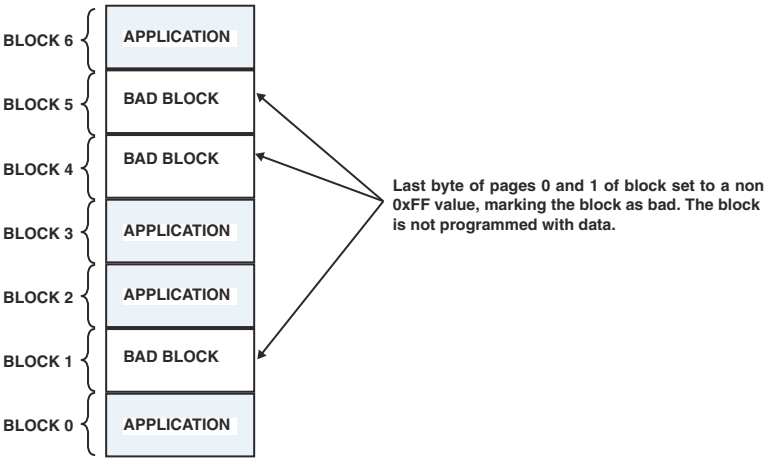


Figure 17-37. Block Skip Mode Typical Usage Scenario

Multiple Image Mode

This mode is enabled with `uwBlockSkipFeature` = 2. Multiple Image Mode allows for multiple copies of the boot stream to be loaded to the NAND flash providing maximum reliability. The number of blocks between each copy of the boot stream is defined by `uwBlockModifier`.

Upon detection of an access to a new block—as in Block Skip Mode, the last byte of the spare area of the first and second page of the block are checked to see if either indicate that the block is bad. If the block is bad, the block offset is applied to the requested data address to fetch from the next copy of the application.

This mode is the only mode that can handle uncorrectable errors from error detection and correction. If an uncorrectable error is received in any block (including block 0), or an error is detected in the parity data, the kernel will fetch the same block of data from the next copy of the application.

The parameter `uwMaxCopies` specifies how many copies of the application are located in the NAND flash. If the processor is booting from the final copy and an uncorrectable error, error in the ECC parity data, or a bad block occurs —the processor enters a safe idle state and the booting process is terminated. This boot method provides greater reliability when regular boot stream updates are expected throughout the life of the product.

Figure 17-38 shows a typical usage scenario.

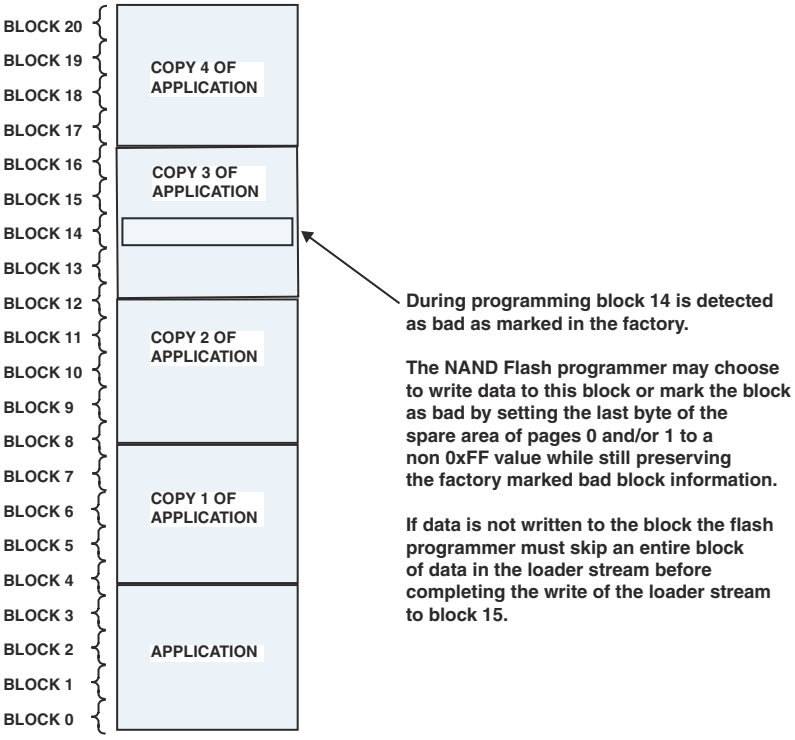


Figure 17-38. Mirror Image Mode Typical Usage Scenario

Reset and Booting Registers

Two registers are used for reset and booting—the software reset register (SWRST) and the system reset configuration register (SYSCR).

Software Reset (SWRST) Register

A software reset can be initiated by setting bits [2:0] in the system software reset field in the software reset register (SWRST) shown in Figure 17-39 on page 17-104. Bit 3 can be used to generate reset upon core-double-fault. A core-double-fault resets both the core and the peripherals, but not the RTC block and most of the DPMC. Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 indicates the software watchdog timer has generated the software reset. Bit 13 indicates the core-double-fault has generated the software reset. Bits [15:13] are read-only and cleared when the register is read. Reading the SWRST also clears bits [15:13] in the SYSCR register. Bits [3:0] are read/write.

Only writing to bits[2:0], resets only the modules in the SCLK domain. It does not clear the core. The program executes normally at the instruction after the MMR write to SWRST. The system is kept in the reset state as long as the bits[2:0] are set to b#111. To release reset, write a zero again. An example is shown in Listing 17-3 on page 17-146. It is not recommended that this functionality be used directly. Rather, call the ROM function bfrom_SysControl() to perform a system reset.

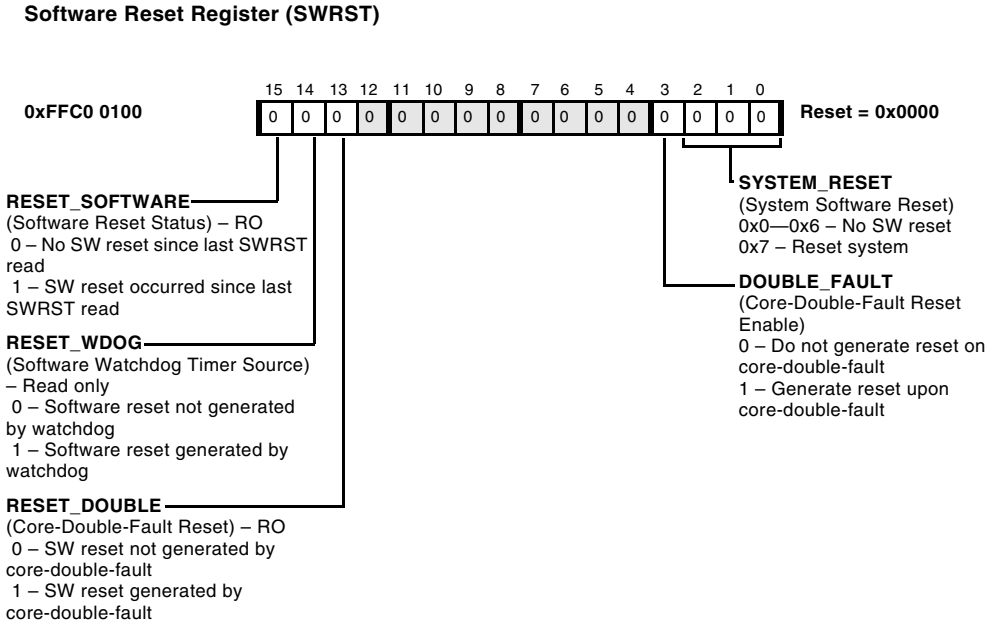


Figure 17-39. Software Reset Register

System Reset Configuration (SYSCR) Register

The values sensed from the BMODE[3:0] pins are mirrored into the system reset configuration register (SYSCR). The values are available for software access and modification after the hardware reset sequence. Software can modify only bits[7:4] in this register to customize boot processing upon a software reset.

The bits [15:13] are exact copies of the same bits in the SWRST register. Unlike the SWRST register, SYSCR can be read without clearing these bits. Reading SWRST also causes SYSCR[15:13] to clear.

The WURESET indicates whether there was a wake up from hibernate since the last hardware reset. The bit cannot be cleared by software.

Bits [11:8] have no booting or reset purpose. These bits control the DMA arbitration.

The software reset configuration register (SYSCR) is shown in Figure 17-40 on page 17-106.

System Reset Configuration Register (SYSCR)

X – state is initialized from BMODE pins during hardware reset

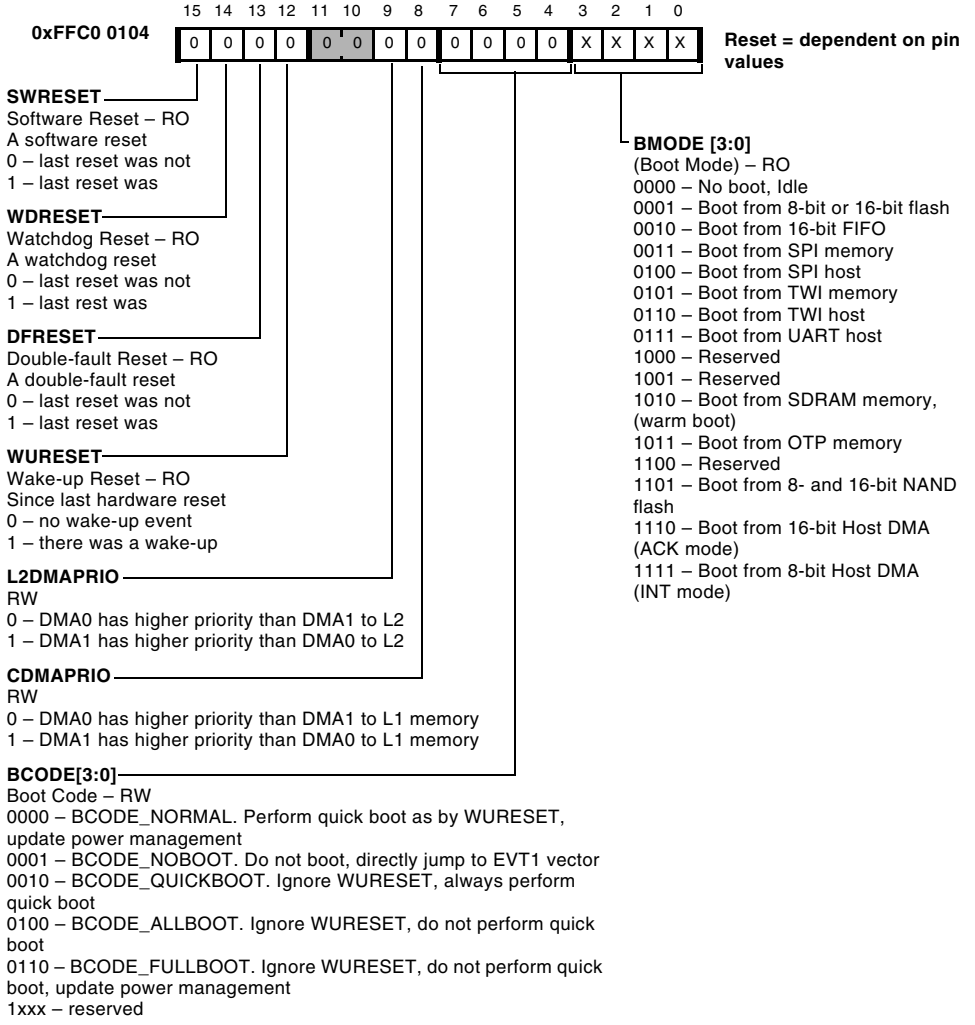


Figure 17-40. System Reset Configuration Register

Boot Code Revision Control (BK_REVISION)

The boot ROM reserves the 32-bits at address 0xEF00 0040 for a four byte version code as shown in Figure 17-41.

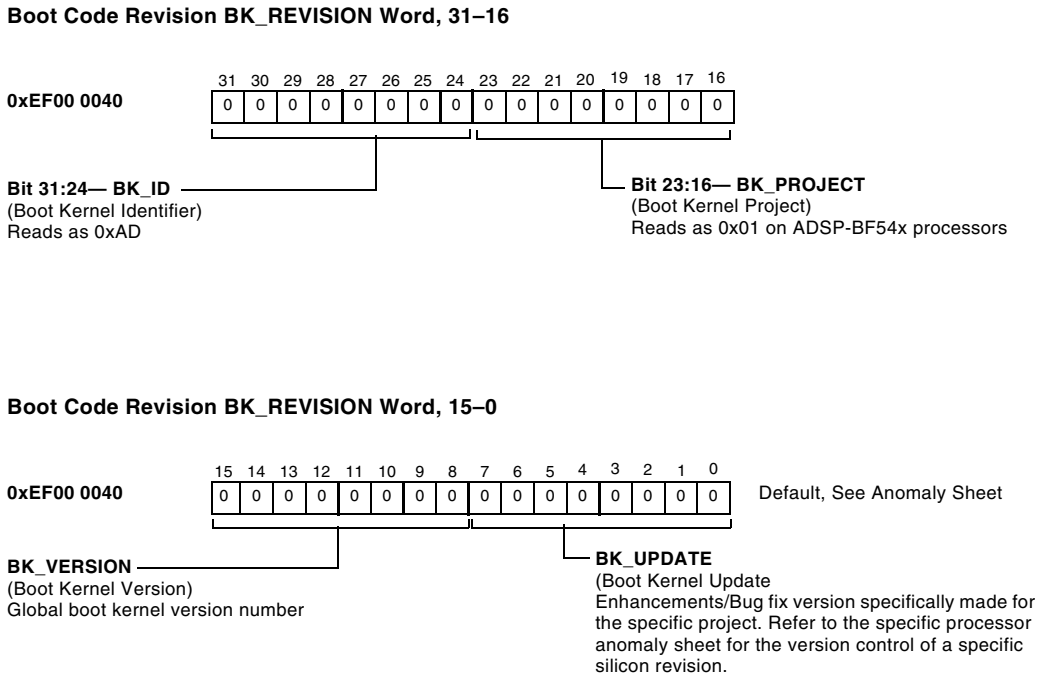


Figure 17-41. Boot Code Revision Code

Boot Code Date Code (BK_DATECODE)

The boot ROM reserves the 32-bits at address 0xEF00 0050 for the build date as shown in Figure 17-42.

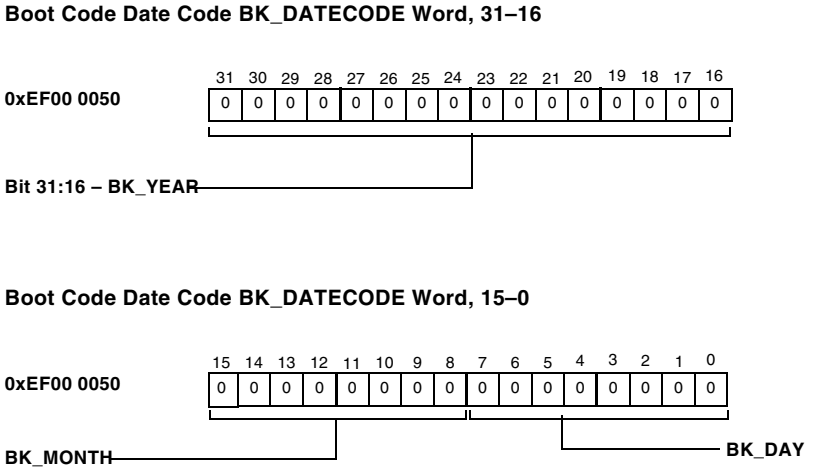


Figure 17-42. Boot Code Date Code

Zero Word (BK_ZEROS)

The boot ROM reserves the 32-bits at address 0xEF00 0048 which always reads as 0x0000 000 as shown in Figure 17-43.

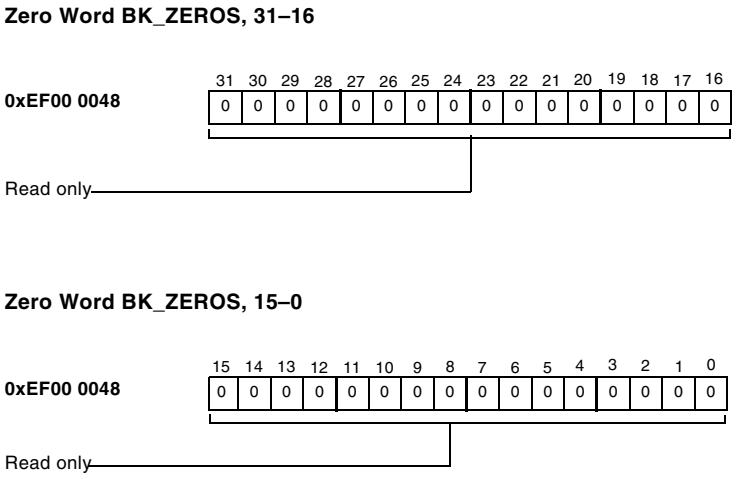


Figure 17-43. Zero Word

Ones Word (BK_ONES)

The boot ROM reserves the 32-bits at address 0xEF00 004C which always reads 0xFFFF FFFF as shown in Figure 17-44.

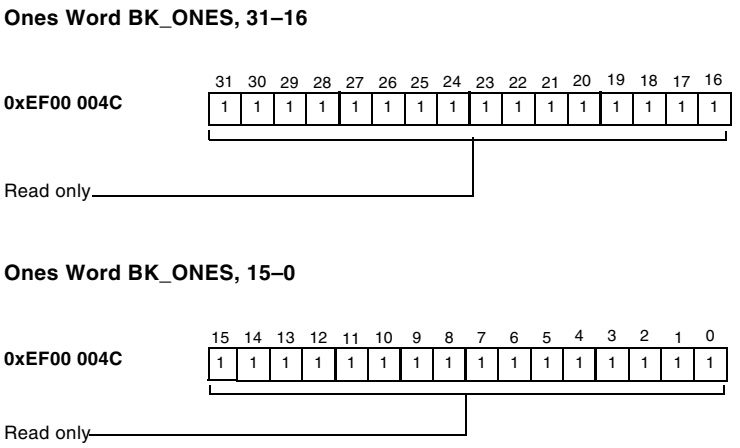


Figure 17-44. Ones Word

OTP Memory Pages for Booting

The following sections describe OTP memory pages for booting.

Lower PBS00 Half Page

The 64-bit lower half of page 0x18 is always read by the preboot routine. These control bits customize the boot process and instruct the preboot routine whether to process further pages and whether the PLL settings have to be changed. Other bits customize the SPI and TWI master boot speed.

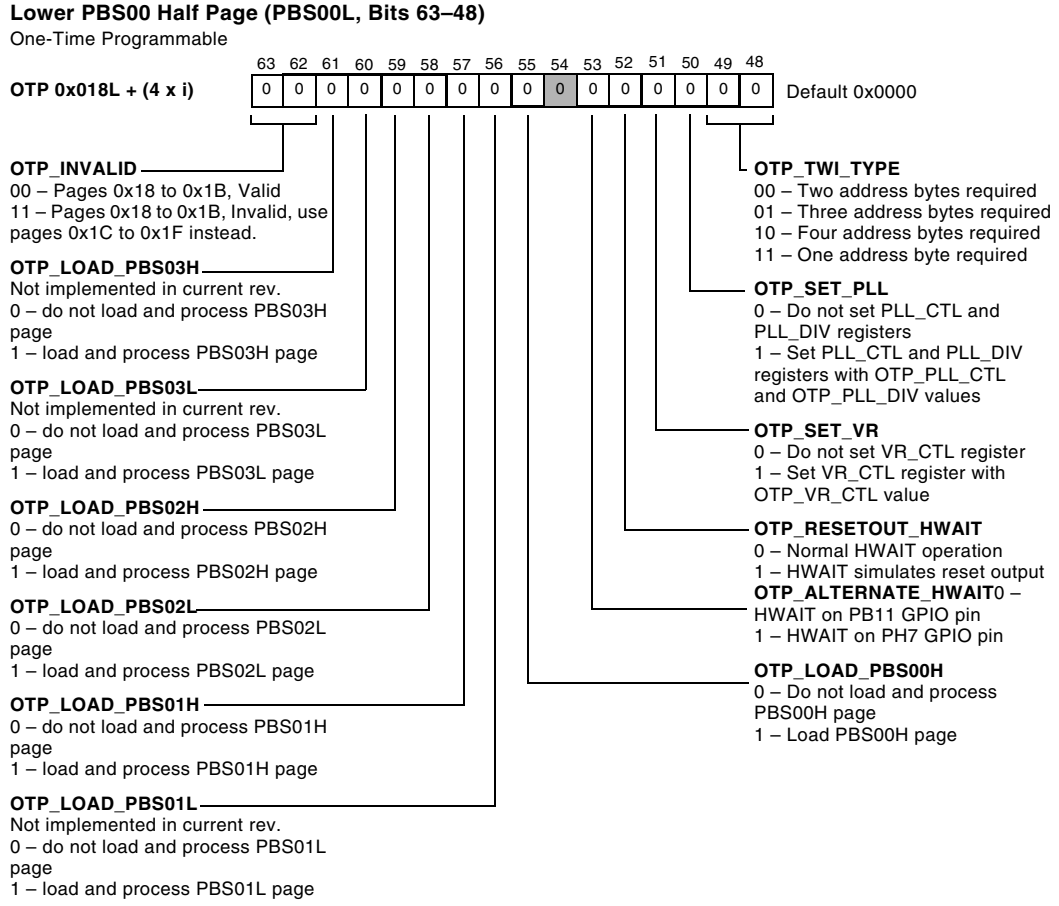


Figure 17-45. Lower PBS00 Half Page (PBS00L, Bits 63–48)

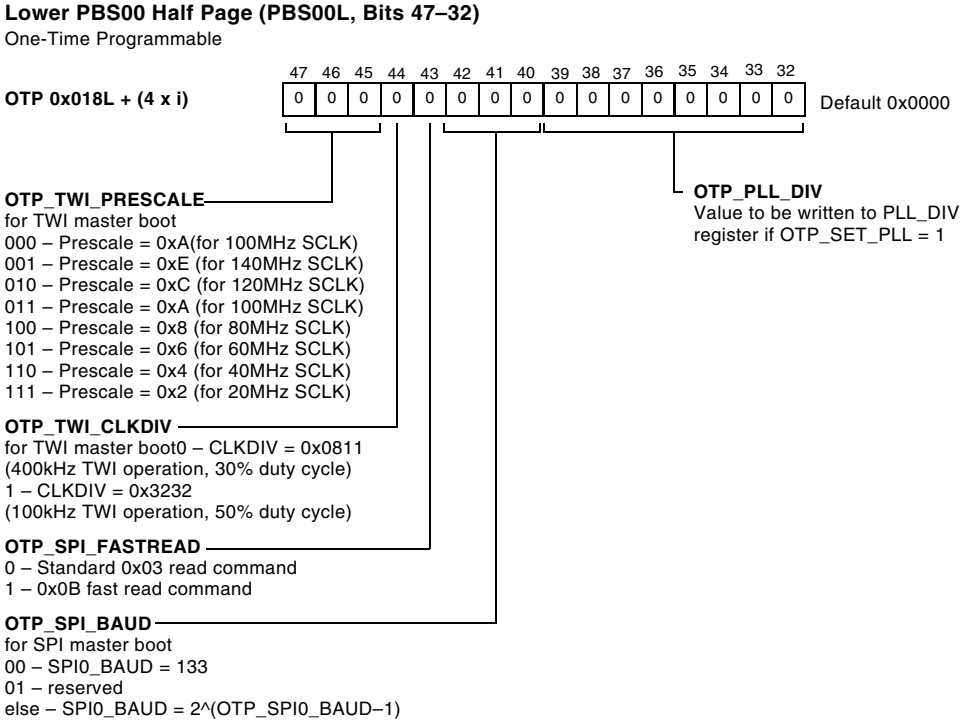


Figure 17-46. Lower PBS00 Half Page (PBS00L, Bits 47–32)

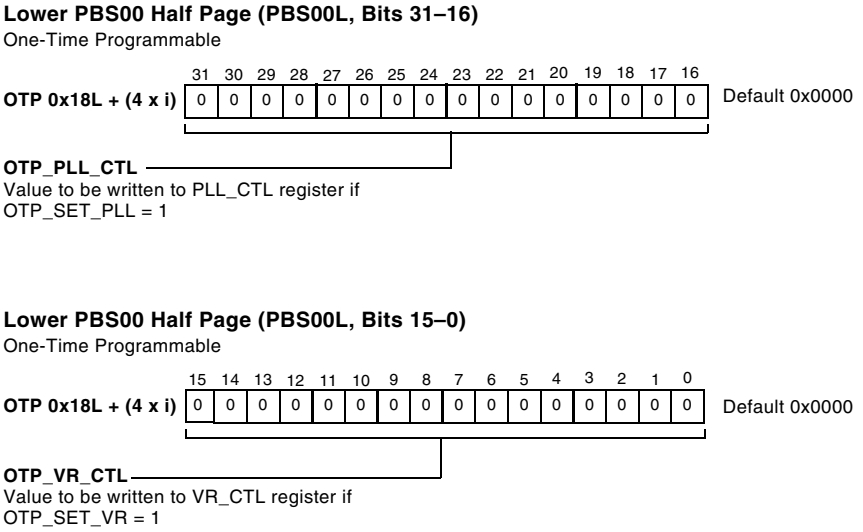


Figure 17-47. Lower PBS00 Half Page (PBS00L, Bits 31–0)

Upper PBS00 Half Page

The preboot routine loads the upper 64-bit half of page PBS00 only if the OTP_LOAD_PBS00H bit in the PBS00L page is set. Page PBS00H customizes the default setting of the asynchronous portion of the EBIU controller.

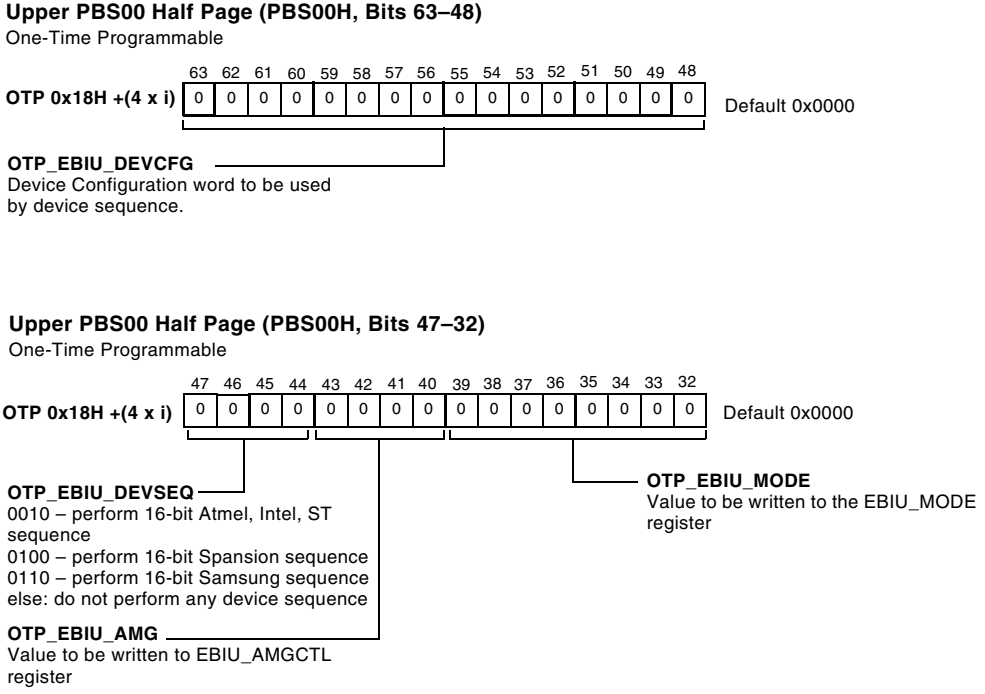


Figure 17-48. Upper PBS00 Half Page (PBS00H, Bits 63–32)

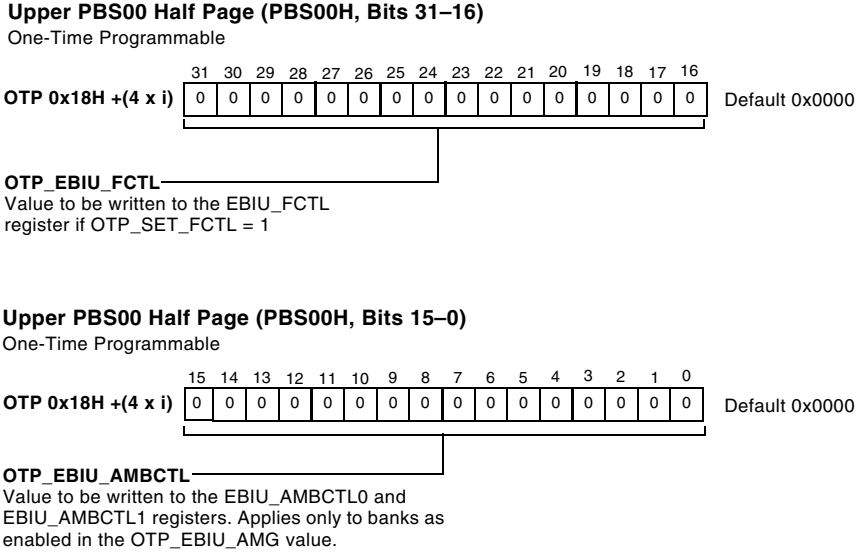


Figure 17-49. Upper PBS00 Half Page (PBS00H, Bits 31–0)

Lower PBS01 Half Page

The half page `PBS01L` is reserved and not used in the current silicon.

Do not use this page as it may be populated in future silicon revisions.

Upper PBS01 Half Page

The preboot routine loads the upper 64-bit half of page 0x19 only if the OTP_LOAD_PBS01H bit in the PBS00L page is set. This page allows the user to disable boot modes. If a disabled boot mode configuration is chosen by the BMODE[3:0] pins, the boot kernel goes into idle state. This half page also provides customization of the NAND flash controller. In OTP boot mode, this pages determines where in OTP memory the boot stream resides.

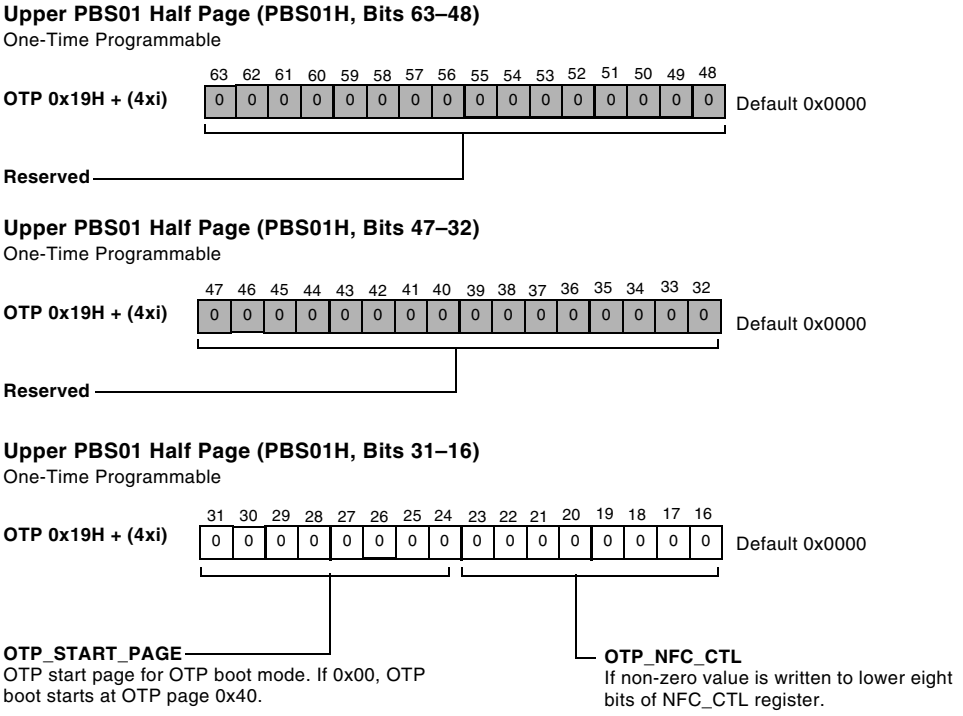


Figure 17-50. OTP Half Page (PBS01H, Bits 63–16)

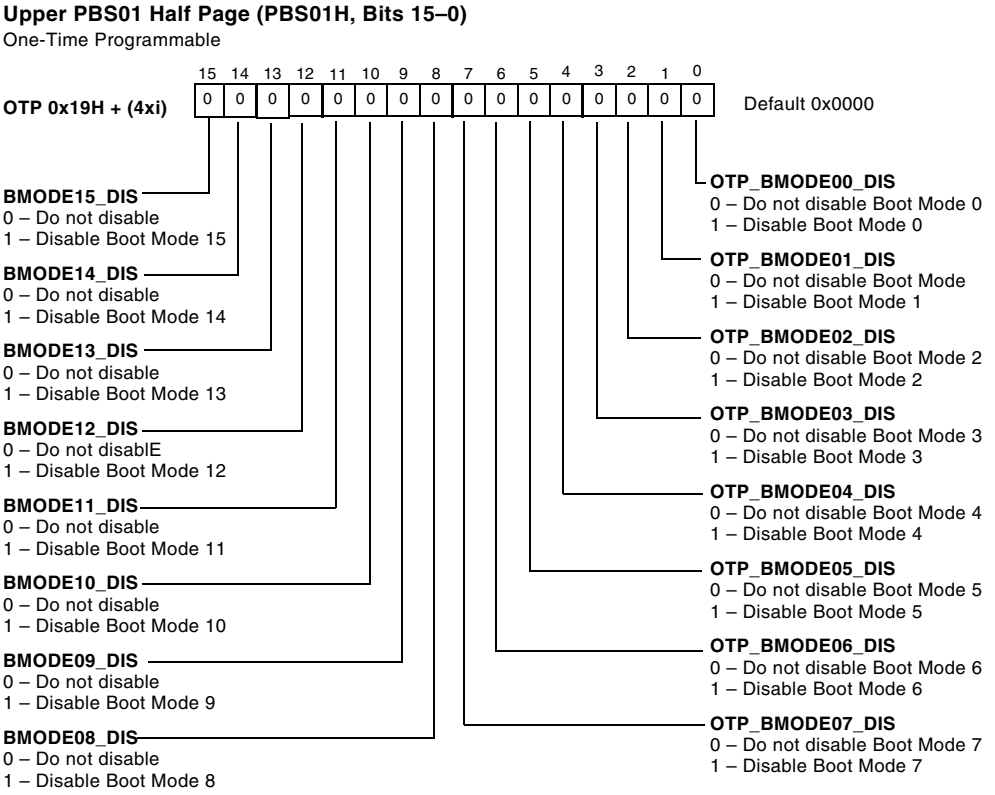


Figure 17-51. OTP Half Page PBS01H (PBS01H, Bits 15–0)

Lower PBS02 Half Page

The preboot routine loads the lower 64-bit half of page 0x1A only if the OTP_LOAD_PBS02L bit in half page PBS00L is set. Half pages PBS02L and PBS02H customize the SDRAM controller settings.

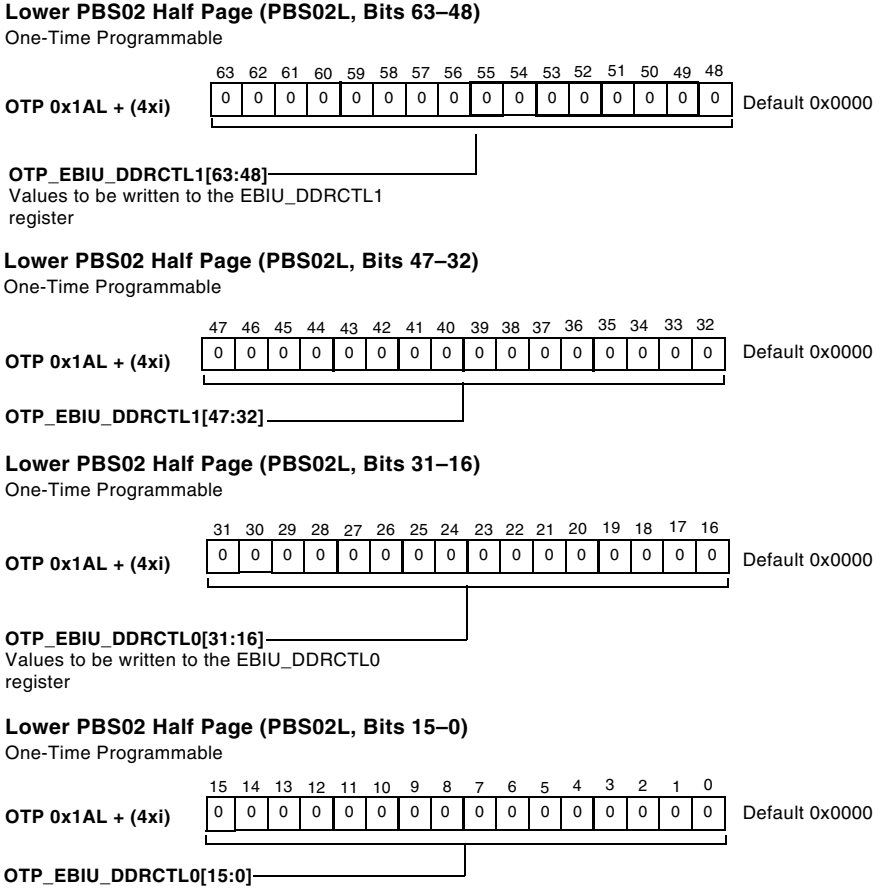


Figure 17-52. Lower PBS02 Half Page (PBS02L, Bits 63–0)

Upper PBS02 Half Page

The preboot routine loads the upper 64-bit half of page 0x16 only if the OTP_LOAD_PBS02H bit in the PBS00L page is set. Half pages PBS02L and PBS02H customize the SDRAM controller settings.

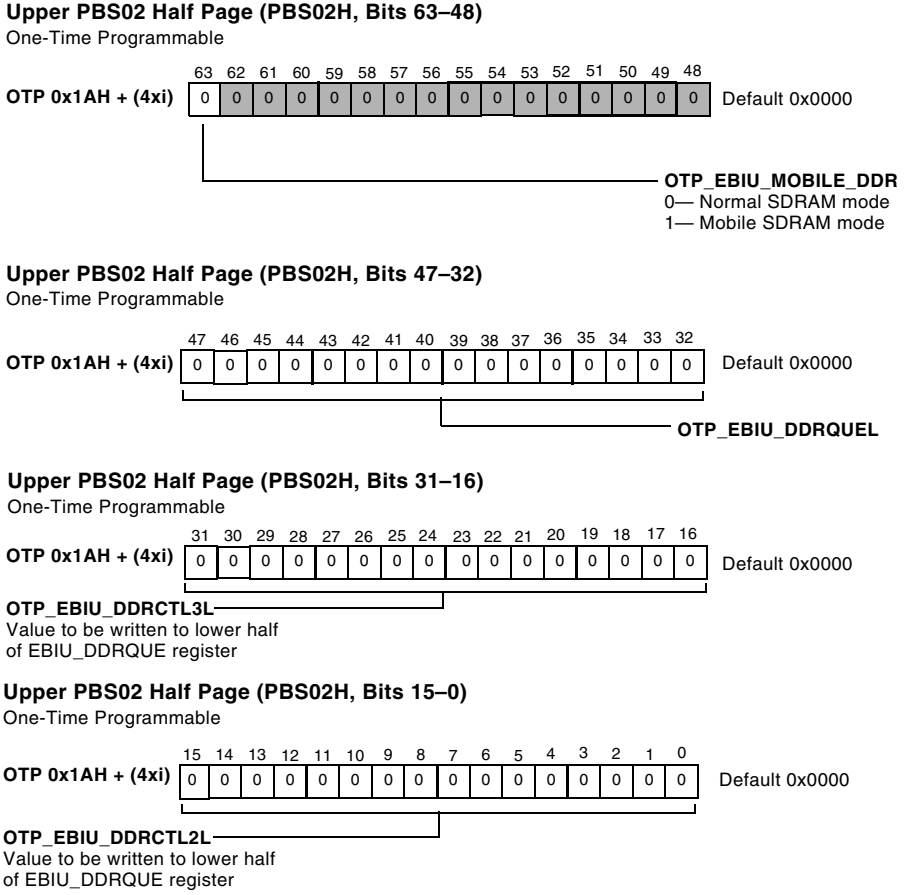


Figure 17-53. Upper PBS02 Half Page (PBS02H, Bits 63–0)

Reserved Half Pages

The half pages PBS01L, PBS03L and PBS03H are reserved and not used in the current silicon.

Do not use these pages as they may be populated in future silicon revisions.

Data Structures

The boot kernel uses specific data structures for internal processing. Advanced users can customize the booting process by changing the content of the structure within the initcode routines. This section uses C language definitions for documentation purposes. Developers can use these structures directly in assembly programs by using the .IMPORT directive. The structures are supplied by the bfrom.h header file in your CCES or VisualDSP++ installation directory.

ADI_BOOT_HEADER

```
typedef struct     {
     s32  dBlockCode;
     void* pTargetAddress;
     s32  dByteCount;
     s32  dArgument;
}     ADI_BOOT_HEADER;
```

The structure ADI_BOOT_HEADER is used by the boot kernel to load and process a block header.

Every block header is loaded to L1 data memory location 0xFF80 7FF0–0xFF80 7FFF first or where pHeader points to. There it is analyzed by the boot kernel.

ADI_BOOT_BUFFER

```
typedef struct     {
     void*  pSource;
   s32    dByteCount;
}     ADI_BOOT_BUFFER;
```

The structure `ADI_BOOT_BUFFER` is used for any kind of buffer. For the user, this structure is important when implementing advanced callback mechanisms.

ADI_BOOT_DATA

```
typedef struct {
     void*  pSource;
     void*  pDestination;
     s16*   pControlRegister;
     s16*   pDmaControlRegister;
     s32    dControlValue;
     s32    dByteCount;
     s32    dFlags;
     s16    uwDataWidth;
     s16    uwSrcModifyMult;
     s16    uwDstModifyMult;
     s16    uwHwait;
     s16    uwSsel;
     s16    uwUserShort;
     s32    dUserLong;
     s32    dReserved;
     ADI_BOOT_ERROR_FUNC*  pErrorFunction;
     ADI_BOOT_LOAD_FUNC*  pLoadFunction;
     ADI_BOOT_CALLBACK_FUNC*  pCallBackFunction;
     ADI_BOOT_HEADER* pHeader;
     void*  pTempBuffer;
     void*  pTempCurrent;
     s32    dTempByteCount;
     s32    dBlockCount;
   s32    dClock;
     void*  pLogBuffer;
     void*  pLogCurrent;
     s32    dLogByteCount;
}     ADI_BOOT_DATA;
```

The structure ADI_BOOT_DATA is the main data structure. A pointer to a ADI_BOOT_DATA structure is passed to most complex subroutines, including load functions, initcode, and callback routines. The structure has two parts. While the first is closely related to internal memory load routines, the second provides access to global boot settings.

Table 17-17 on page 17-122 describes the data structures.

Table 17-17. Structure Variables, ADI_BOOT_DATA

Variable	Description
pSource	In the context of the boot kernel, the pSource pointer points either to the start address of the entire boot stream or to the header of the next boot block. In the context of memory load routines pSource points to the source address of the DMA work unit.
pDestination	The pDestination pointer is only used in memory load routines. It points to the destination address of the DMA work unit. It points to either 0xFF80 7FF0 when a header is loaded, or the target address when the payload data is loaded.
pControlRegister	This pointer holds the MMR address of the peripheral's main control register (for example UARTx_LCR or SPIx_CTL)
pDmaControlRegister	This pointer holds the MMR address of the DMAx_CONFIG register for the DMA channel in use.
dControlValue	The lower 16 bits of this value are written to the pControlRegister location each time a DMA work unit is started.
dByteCount	Number of bytes to be transferred.
dFlags	The lower 16 bits of this variable hold the lower 16 bits of the current block code. The upper 16 bits hold global flags. See "dFlags Word" on page 17-125.
uwDataWidth	This instructs the memory load routine to use: 0 – 8-bit DMA 1 – 16-bit DMA 2 – 32-bit DMA
uwSrcModifyMult	This is the multiplication factor used by the DMA source. A value of 1 sets the source modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.

Table 17-17. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
uwDstModifyMult	This is the multiplication factor used by the DMA destination. A value of 1 sets the destination modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwHwait	This 16-bit value holds the GPIO used for HWAIT signaling. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port. For example, GPIO PH11 has a value of 0x080B, PB7 has a value of 0x0207, PG0 has a value of 0x0700.
uwSsel	This 16-bit value holds the GPIO used for SPI slave select. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.
uwUserShort	The programmer can use this 16-bit value for passing parameters between modules of a customized booting scheme.
dUserLong	The programmer can use this 32-bit value for passing parameters between modules of a customized booting scheme.
dReserved	This 32-bit value is reserved for future development.
pErrorFunction	This is the pointer to the error handler. See "Error Handler" on page 17-48.
pLoadFunction	This is the pointer to the function responsible for loading data. See "Load Functions" on page 17-49
pCallBackFunction;	This is the pointer to the callback function. See "Callback Routines" on page 17-45
pHeader	The pHeader pointer holds the address for intermediate storage of the block header. By default this value is set to 0xFF80 7FF0.
pTempBuffer	This pointer tells the boot kernel what memory to use for intermediate storage when the BFLAG_INDIRECT flag is set for a given block. The pointer defaults to 0xFF90 7E00. The value can be modified by the initcode routine, but there would be some impact to the CCES or VisualDSP++ tools.
pTempCurrent	Defaults to the pTempBuffer value. A load function can modify this value to manipulate subsequent callback and memory DMA routines.

Table 17-17. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
dTempByteCount	This is the size of the intermediate storage buffer used when the BFLAG_INDIRECT flag is set for a given block. This value defaults to 256 and can be modified by an initcode routine. When increasing this value, the pTempBuffer must also be changed since by default the block is at the end of a physical data memory block.
dBlockCount	This 32-bit variable counts the boot blocks that are processed by the boot kernel. If the user sets this value to a negative value, the boot kernel exits when the variable increments to zero.
dClock	The dClock variable holds information about the clock divider used by individual (serial) boot modes.
pLogBuffer	Pointer to the circular log buffer. By default the log buffer resides in L1 scratch pad memory at address 0xFFB0 0400.
pLogCurrent	Pointer to the next free entry of the circular log buffer.
dLogByteCount	Size of the circular log buffer, default is 0x400 bytes.

dFlags Word

Figure 17-55 and Figure 17-54 on page 17-125 describe the `dFlags` word. `dFlags [15-0]` is a copy of Block Code[15–0] of the block currently being processed.

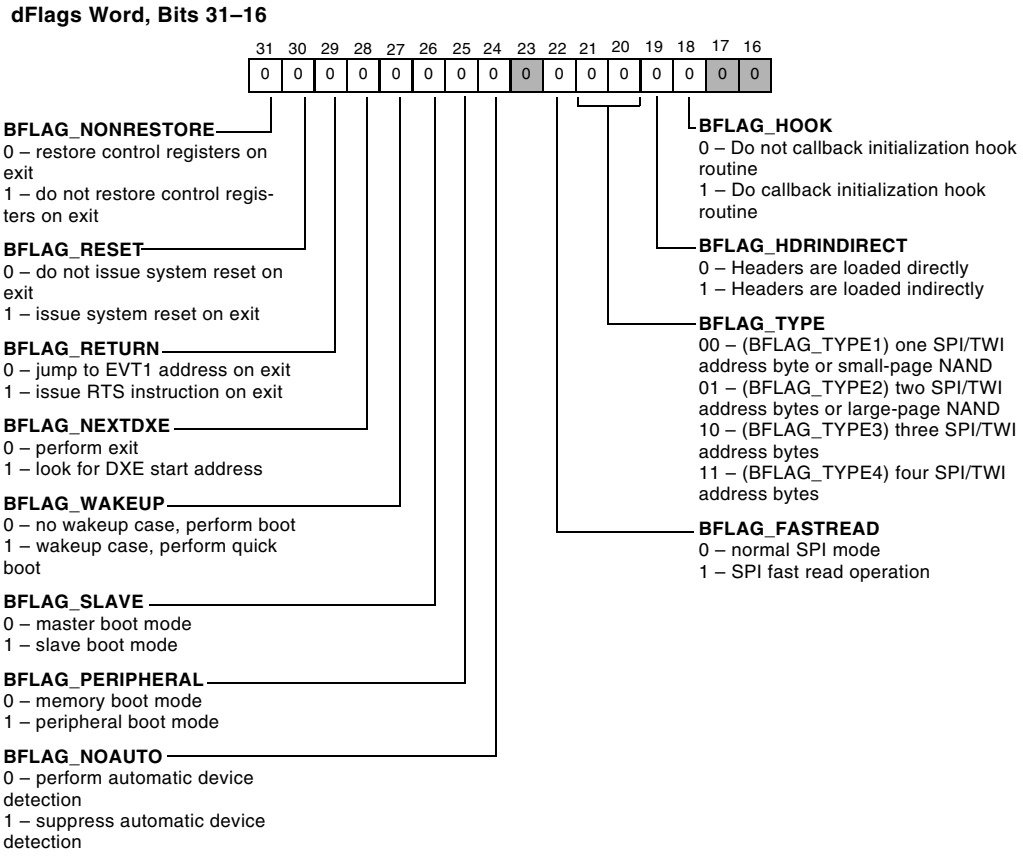


Figure 17-54. dFlags Word (Bits 31–16)

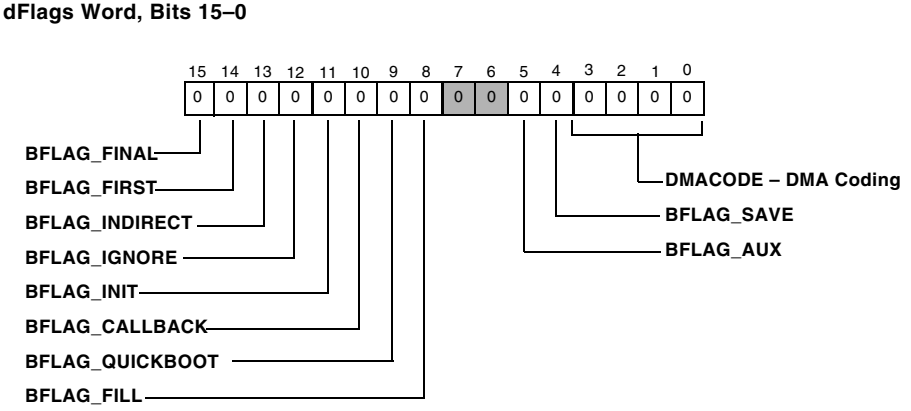


Figure 17-55. dFlags Word (Bits 15–0)

ADI_BOOT_NAND

```
typedef struct{
    ADI_BOOT_NAND_DEVICE DeviceInfo;
    ADI_BOOT_NAND_BUFFER MainBuffer;
    ADI_BOOT_NAND_BUFFER PrefetchBuffer;
    ADI_BOOT_NAND_ACCESS AddressRequested;
    ADI_BOOT_NAND_ADDRESS AddressCycles;
    ADI_BOOT_NAND_ECC EccParity;
    ADI_BOOT_DATA *pBootData;
void *pReserved;
} ADI_BOOT_NAND;
```

The boot kernel uses a number of data structures for internal processing. Advanced users may manipulate some of contents of the structures from within initcode routines to customize the boot process further.

ADI_BOOT_NAND is the central structure and is used only by the NAND flash boot kernel. The pointer to ADI_BOOT_NAND is stored in the dUserLong parameter of ADI_BOOT_DATA when NAND flash boot mode is enabled. This pointer provides access to the ADI_BOOT_NAND structure through initialization routines to further customize the booting process.

Table 17-18. Structure Variables, ADI_BOOT_NAND

Variable	Description
DeviceInfo	Properties relating to the NAND flash device
MainBuffer	Information relating to the current contents of the MainBuffer.
PrefetchBuffer	Information relating to the current contents of the PrefetchBuffer.
AddressRequested	Details of the requested address when the address is converted to an address suitable for accessing the NAND flash.
AddressCycles	Information required to correctly read from the NAND flash device.
EccParity	Stores the error correction parity data for a NAND flash page and controls the operating mode of the NAND flash boot kernel.
pBootData	Pointer to the global ADI_BOOT_DATA structure.
pReserved	Reserved for future enhancements. Do not use.

ADI_BOOT_NAND_DEVICE

```
typedef struct{
    u32  udIdCode;
    u32  udIdType;
    u16  uwBusWidth;
    u16  uwColumnMaskCount;
    u32  udColumnMask;
    u16  uwPageMaskCount;
    u32  udPageMask;
    u16  uwSpareMaskCount;
    u16  uwSpareAreaBit;
    u32  udBlockSize;
    u16  uwPageSize;
    u16  uwPagesPerBlock;
    u16  uwSpareAreaSize;
    u16  uwSpareAreaModifier;
    u16  uwNFCPages;
} ADI_BOOT_NAND_DEVICE;
```

This structure provides details about the NAND flash device connected to the NFC. For booting from supported small-page NAND flash devices not all parameters are used and thus initialized. For supported large-page NAND flash memories, the structure is initialized after reading the electronic signature of the device. The fourth byte of the four byte electronic signature contains information for initialization of the entire structure.

Table 17-19. Structure Variables, ADI_BOOT_NAND_DEVICE

Variable	Description
udIdCode	The electronic signature of the device as received after issuing the Read Electronic Signature command. This is only used for large-page NAND flash devices. It is not populated if a small-page device is detected, since only a single small-page type is supported.
udType	0 indicates a small-page device. 1 indicates a large-page device.
uwBusWidth	Bus width of the device. '0' for 8-bit. '1' for 16-bit.
uwColumnMaskCount	Number of bits required to address all columns within a NAND flash page (excluding the spare area). This is used to translate the address pSource in ADI_BOOT_DATA to the format required for addressing the NAND flash device.
udColumnMask	Used to extract the column within a page being addressed from the requested source address.
uwPageMaskCount	Number of bits required to address all pages within a single NAND flash block.
udPageMask	Used to extract the page number within a block being addressed from the source address.
uwSpareMaskCount	Number of bits required to address all columns within the spare area at the end of a NAND flash page.
uwSpareAreaBit	Contains the bit position to be set to address the spare area of the NAND flash page.
udBlockSize	Block size of the device in bytes (excluding the spare area).
uwPageSize	Page size of the device in bytes (excluding the spare area).
uwPagesPerBlock	Number of pages within a block.
uwSpareAreaSize	Number of bytes within the spare area of a page.
uwSpareAreaModifier	Number of bytes in the spare area dedicated to each 256 byte NAND flash controller page.
uwNFCPages	Number of 256 byte NAND Flash controller pages within a full NAND flash page.

ADI_BOOT_NAND_BUFFER

```
typedef struct{
    void * pBegin;
    u16  uwLoadedNFCPage;
    u16  uwLoadedNANDPage;
    u16  uwLoadedNANDBlock;
} ADI_BOOT_NAND_BUFFER;
```

The ADI_BOOT_NAND_BUFFER structure provides details of the current contents of a 256 byte buffer. There are two of these buffers required for NAND flash boot. The buffer provides details on the location of the buffer as well as its current contents. Since 256 byte blocks of data are read from the NAND flash memory at a time, the kernel can determine if a new data fetch is required from the NAND flash or whether the data resides in one of the two buffers located in internal memory.

Table 17-20. Structure Variables, ADI_BOOT_NAND_BUFFER

Variable	Description
pBegin	Pointer to the first address of a 256 byte buffer.
uwnLoadedNFCPage	The currently loaded 256 byte NAND flash controller sub-page.
uwLoadedNANDPage	The currently loaded NAND flash page.
uwLoadedNANDBlock	The currently loaded NAND flash block.

ADI_BOOT_NAND_ACCESS

```
typedef struct{
    u16  uwAccessNFCPage;
    u16  uwAccessNANDPage;
    u16  uwAccessNANDBlock;
} ADI_BOOT_NAND_ACCESS;
```

The actual page and block in which the data resides can be calculated from the source address provided by the main kernel and the contents of the ADI_BOOT_NAND_DEVICE structure. This structure is also used along with the ADI_BOOT_NAND_BUFFER to determine if data needs to be fetched from the NAND flash memory or whether it already resides in internal memory.

Table 17-21. Structure Variables, ADI_BOOT_NAND_ACCESS

Variable	Description
uwAccessNFCPage	The requested 256 byte NAND flash controller sub-page to be accessed.
uwAccessNANDPage	The requested NAND flash page to be accessed.
uwAccessNANDBlock	The requested NAND flash block to be accessed.

ADI_BOOT_NAND_ADDRESS

```
typedef struct{
void *pSource;
u32 udMainOffset;
u32 udPrefetchOffset;
u16 uwNumAddressCycles;
u16 uwNumCommands;
u16 uwSerialAccess;
ADI_BOOT_NAND *pNandInfo
#pragma align 4
u8 ubCommand0;
u8 ubAddress0;
u8 ubAddress1;
u8 ubAddress2;
u8 ubAddress3;
u8 ubAddress4;
u8 ubCommand1;
} ADI_BOOT_NAND_ADDRESS;
```

`ADI_BOOT_NAND_ADDRESS` is modified when the NAND flash boot kernel decodes the source address provided by the main kernel. When a booting feature is used that detects bad blocks or uncorrectable errors, offsets for addressing alternative blocks are applied. When the address is decoded, the structure is filled with the NAND flash controller commands and address cycles needed for retrieving the required data.

For supported small-page NAND flash devices, the number of address cycles is always four and the number of command cycles is one. For large-page NAND flash devices, the default number of address cycles is five. Since the upper addressing boundaries of the NAND flash device cannot be determined from the electronic signature, the kernel is unable to calculate the exact number of address cycles required to perform a read from the NAND flash. A majority of large-page NAND flash devices simply ignore any address cycles on a page read command that are not required. If a NAND flash device is not capable of ignoring the additional address cycles and it requires less than the default five address cycles for a page read operation then the device cannot be supported for NAND boot

functionality. To remove the redundant address cycles, the required number of address cycles can be reconfigured within an initialization file executed before loading the main application.

Table 17-22. Structure Variables, ADI_BOOT_NAND_ADDRESS

Variable	Description
pSource	The source address to be accessed.
udMainOffset	The current block offset applied to data loaded into the main buffer.
udPrefetchOffset	The current block offset applied to data loaded into the prefetch buffer.
uwNumAddress-Cycles	The number of address cycles required to access the NAND flash device. This is set to 4 for small-page device booting and 5 for large-page devices.
uwNumCommands	The number of command cycles required to perform a read access from the NAND flash device. This parameter is set to 1 for small-page devices and 2 for large-page devices.
uwSerialAccess	Indicates that the next read access is from the next sequential 256 byte page to the previous access. This allows for the removal of the issuing of a read transaction thus optimizing throughput without waiting on unnecessary ready/$\overline{\text{busy}}$ assertions.
pNandInfo	Pointer to ADI_BOOT_NAND structure
ubCommand0	The first command to be issued to perform a page read from the NAND flash device.
ubAddress0	The first address cycle issued when performing a page read command.
ubAddress1	The second address cycle issued when performing a page read command.
ubAddress2	The third address cycle issued when performing a page read command.
ubAddress3	The fourth address cycle issued when performing a page read command.
ubAddress4	The fifth address cycle issued when performing a page read command.
ubCommand1	The second command to be issued to perform a page read from the NAND flash device. Only used for large-page devices.

ADI_BOOT_NAND_ECC

```
typedef struct{
#pragma align 4
u16 uwIndex;
u32 udNFCParity[32];
u16 uwError;
u16 uwBlockSkipFeature;
u16 uwBlockModifier;
u16 uwMaxCopies;
u16 uwCurrentCopy;
} ADI_BOOT_NAND_ECC;
```

This structure provides stack storage for the error correction parity data read from the spare area of a page when an access to a new NAND flash page is detected. The spare area contains parity data for each 256 byte block in a page. This allows for error correction and detection to be performed on every 256 byte load from the NAND flash. Enough storage space is provided to support devices up to and including a page size of 8K bytes. `ADI_BOOT_NAND_ECC` also contains the fields that need to be modified to enable the NAND flash boot options that skip bad blocks or boot from mirror images of the original boot stream located in other memory blocks.

Table 17-23. Structure Variables, ADI_BOOT_NAND_ECC

Variable	Description
nIndex	Index used to access the udNFCParity array
udNFCParity	A 32 deep long word array providing storage for up to 32 256-byte NAND Flash Controller error correction parity data. The array provides support for page sizes up to and including 8 Kbytes.
uwError	Error that was generated within the error correction routine. 0 – No Error 1 – Error found in parity data 2 – Uncorrectable error
uwBlockSkipFeature	Specifies the NAND flash boot technique to be implemented. Defaults to 0 unless otherwise altered through an initialization sequence. 0 – Sequential booting from a single boot stream. No bad block checking performed. 1 – Block Skip Method, allowing for a single boot stream loaded to the NAND flash to skip bad blocks. 2 – Mirror Image Mode, allowing for booting from multiple copies of the application in the event that an uncorrectable error or error in the ECC parity data is detected.
uwError	Indicates the error returned from the error correction routine if one occurred. 0 – No error or correctable error. 1 – Error in ECC parity data. 2 – Uncorrectable error.
uwBlockModifier	The number of blocks to skip if a bad block is detected. If uwBlockSkipFeature is 0 this value is ignored. For an uwBlockSkipFeature value of 1 this parameter must be 1. For an uwBlockSkipFeature of 2 this parameter may be any value indicating the number of blocks between multiple copies of the application.
uwMaxCopies	The number of copies of the application stored in the NAND flash device. Only applicable if uwBlockSkipFeature is 2.
uwCurrentCopy	Indicates the current copy of the application that is being accessed. Only applicable if uwBlockSkipFeature is 2.

Callable ROM Functions for Booting

The following functions support boot management.

BFROM_FINALINIT

Entry address: 0xEF00 0002

Arguments: no arguments

C prototype: `void bfrom_FinalInit (void);`

The `bfrom_FinalInit` function never returns. It only executes a `JUMP` to the address stored in `EVT1`.

BFROM_PDMA

Entry address: 0xEF00 0004

Arguments: pointer to `ADI_BOOT_DATA` in `R0`

C prototype: `void bfrom_PDma (ADI_BOOT_DATA *p);`

This is the load function for peripherals such as SPI and UART that support DMA in their boot modes.

BFROM_MDMA

Entry address: 0xEF00 0006

Arguments: pointer to `ADI_BOOT_DATA` in `R0`

C prototype: `void bfrom_MDma (ADI_BOOT_DATA *p);`

This is the load function used for memory boot modes including the FIFO mode. This routine is also reused when the `BFLAG_FILL` or the `BFLAG_INDIRECT` flags are specified.

BFROM_MEMBOOT

Entry address: 0xEF00 0008

Arguments:

pointer to boot stream in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_MemBoot (void* pBootStream, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes any boot stream that maps to the Blackfin memory starting from address pBootStream.

To boot a new application that may overwrite the calling application, the dFlags word is usually zero. When done, the routine does not return, but jumps to the EVT1 vector address. If the BFLAG_RETURN flag is set, an RTS is executed instead and the routine returns to the parent function. In this way, fractions of an application can be loaded.

If the dBlockCount parameter is zero or a positive value, all boot blocks are processed until the BFLAG_FINAL flag is detected. If dBlockCount is a negative value, the negative number represents the number of blocks to be booted. For example, –1 causes the kernel to return immediately, –2 processes only one block.

The routine returns the updated source address pSource of the boot stream (for example, the first unused address after the processed boot stream).

The BFLAG_NEXTDXE flag suppresses boot loading. The boot kernel steps through the boot stream by analyzing the next-DXE pointers (in the ARGUMENT field of a BFLAG_FIRST block) and jumping to the next DXE. Assuming that the boot image is a chained list of boot streams, the boot kernel returns the absolute start address of the requested boot stream. In this example, the start address of the third boot stream (DXE) in a flash device is returned.

```
bfrom_MemBoot((void*)0x20000000,
BFLAG_RETURN|BFLAG_NEXTDXE,-3, NULL);
```

In the above example, the routine would return 0x2000 0000 when dBlockCount was set to –1. If the parameter dBlockCount is zero or positive when used along with the BFLAG_NEXTDXE command, the kernel returns when the BFLAG_FIRST flag on a header in the next-DXE chain is not set.

If the BFLAG_HOOK switch is set, the memboot routine call (pCallHook routine) after the ADI_BOOT_DATA structure is filled with default values. It then can overrule the default settings of the structure.

BFROM_TWIBOOT

Entry address: 0xEF00 000C

Arguments:

TWI address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_TwiBoot (s32 dTwiAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes boot streams residing in TWI memories, using the TWI0 controller. It differs from the `BFROM_MEMBOOT` routine in that some functionality is TWI specific.

Additional bits in the `dFlags` word are relevant. The user should always set the `BFLAG_PERIPHERAL` flag but never the `BFLAG_SLAVE` bit. The `BFLAG_TYPE` tells the boot kernel when addressing mode is required for the TWI memory. The boot kernel derives the values for the `TWI0_CONTROL` and `TWI0_CLKDIV` registers from the lower four bits of the `dFlags` word. See Chapter 23, “Two-Wire Interface Controllers”.

BFROM_SPIBOOT

Entry address: 0xEF00 000A

Arguments:

SPI address in `R0`

`dFlags` in `R1`

`dBlockCount` in `R2`

`pCallHook` passed over the stack in [FP+0x14]

updated block count returned in `R0`

C prototype:

```
s32 bfrom_SpiBoot (s32 dSpiAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This SPI master boot routine processes boot streams residing in SPI memories, using the SPI0 controller. It differs from the `BFROM_TWIBOOT` routine in that some functionality is SPI specific. The fourth argument `pCallHook`

is passed over the stack. It provides a hook to call a callback routine after the ADI_BOOT_DATA structure is filled with default values. For example, the pCallHook routine may overwrite the default value of the uwSsel value in the ADI_BOOT_DATA structure. The coding follows the rules of uwHWAIT (see "Boot Host Wait (HWAIT) Feedback Strobe" on page 17-33). A value of 0x0504 represents GPIO PE4 ($\overline{\text{SPIOSEL1}}$), 0x0505 represents PE5 ($\overline{\text{SPIOSEL2}}$) and so on.

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SLAVE bit. The BFLAG_NOAUTO flag instructs the system to skip the SPI device detection routine. The BFLAG_TYPE then tells the boot kernel what addressing mode is required for the SPI memory. (see "SPI Device Detection Routine" on page 17-71). The BFLAG_FASTREAD flag controls whether standard SPI read (0x3 command) or fast read (0xB) is performed. The boot kernel writes the lower bits of the dFlags word to the SPI0_BAUD registers.

BFROM_OTPBOOT

Entry address: 0xEF00 000E

Arguments:

OTP byte address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

Updated block count returned in R0

C prototype:

```
s32 bfrom_OtpBoot (s32 dOtpAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This OTP boot routine processes boot streams residing in the on-chip, serial OTP memory. Unlike the `bfrom_OtpRead()` function which uses the half-page addressing method, this one requires byte addressing. For example, set the `dOtpAddress` argument to 0x400 to process a boot stream starting from OTP page 0x40. Remember that one OTP page spans 16 bytes.

BFROM_NANDBOOT

Entry address: 0xEF00 0010

Arguments:

NAND Flash address in `R0`

`dFlags` in `R1`

`dBlockCount` in `R2`

`pCallHook` passed over the stack in [FP+0x14]

updated block count returned in `R0`

C prototype:

```
s32 bfrom_NandBoot(s32 dNandAddress,
 s32 dFlags, s32 dBlockCount, ADI_BOOT_HOOK_FUNC *pCallHook)
```

This NAND flash boot routine processes boot streams residing in NAND flash memories, using the NAND Flash Controller. Some functionality is NAND flash specific.

Additional bits in the `dFlags` word are relevant. When the `BFLAG_NOAUTO` flag is set the `BFLAG_TYPE` field is used to indicate whether the connected NAND flash is a small-page or large-page device.

`BFLAG_TYPE` = 00 (`BFLAG_TYPE1`) indicates small-page NAND Flash

`BLAG_TYPE` = 01 (`BFLAG_TYPE2`) indicates large-page NAND Flash

BFLAG_TYPE — values of 11 and 10 are reserved

Detection of a reserved value results in a call to the error handler.

In the event the NFC_CTL register is set to the default reset value of 0x0200 prior to the call to bfrom_NandBoot(), the read and write delay strobes of the NFC_CTL register will each be set to three providing t_{RP} and t_{WP} timings of four SCLK cycles.

BFROM_BOOTKERNEL

Entry address: 0xEF00 0020

Arguments:

pointer to ADI_BOOT_DATA in R0

returns updated source address pSource in R0

C prototype:

```
s32 bfrom_BootKernel (ADI_BOOT_DATA *p);
```

This ROM entry provides access to the raw boot kernel routine. It is the user's responsibility to initialize the items passed in the ADI_BOOT_DATA structure. Pay particular attention that the function pointers (pLoadFunction, and pErrorFunction) point to functional routines.

BFROM_CRC32

Entry address: 0xEF00 0030

Arguments:

pointer to look-up table in R0

pointer to data in R1

dByteCount in R2

initial CRC value in R0

CRC value returned in R0

C prototype:

```
s32 bfrom_Crc32 (s32 *pLut, void *pData,
 s32 dByteCount, s32 dInitial);
```

This routine calculates the CRC32 checksum for a given array of bytes. The look-up table is typically generated by the BFROM_CRC32POLY routine. During the boot process this routine is called by the BFROM_CRC32CALLBACK routine. The dInitial value is normally set to zero unless the CRC32 routine is called in multiple slices. Then, the dInitial parameter expects the result of the former run.

BFROM_CRC32POLY

Entry address: 0xEF00 0032

Arguments:

pointer to look-up table in R0

polynomial in R1

updated block count returned in R0

C prototype:

```
s32 bfrom_Crc32Poly (unsigned s32 *pLut, s32 dPolynomial);
```

This function generates a 1024-byte look-up table from a given CRC polynomial. During the boot process this routine is hidden by the BFROM_CRC32INITCODE routine.

BFROM_CRC32CALLBACK

Entry address: 0xEF00 0034

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

pointer to `ADI_BOOT_BUFFER` in `R1* Callback Flags in R2`

C prototype:

```
s32 bfrom_Crc32Callback (ADI_BOOT_DATA *pBS, ADI_BOOT_BUFFER
*pCS, s32 dCbFlags);
```

This is a wrapper function that ensures the `BFROM_CRC32` subroutine fits into the boot process.

BFROM_CRC32INITCODE

Entry address: 0xEF00 0036

Arguments:
pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_Crc32Initcode (ADI_BOOT_DATA *p);
```

This is an initcode residing in ROM with two jobs:

Register `BFROM_CRC32CALLBACK` as a callback routine to the `pCallback` pointer in `ADI_BOOT_DATA`.

Call `BFROM_CRC32POLY` to generate the look-up table.

This function is unlikely to be called by user code directly. This function is called as an initcode during the boot process when the CRC calculation is desired. See “CRC Checksum Calculation” on page 17-48 for details.

Programming Examples

The following sections provide programming examples for system reset and booting.

System Reset

To perform a system reset, use the code shown in Listing 17-1 or Listing 17-2. As described in the code comments below, the system soft reset takes five system clock cycles to complete, so a delay loop is needed. This code must reside in L1 memory for the system soft reset to work properly.

Listing 17-1. System Reset in Assembly

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

/* Wait for System reset to complete (needs to be 5 SCLKs). */
/* Assuming a worst case CCLK:SCLK ratio (15:1), use 5*15 = 75 */
/* as the loop count. */
P1 = 75;
LSETUP(start, end) LC0 = P1 ;
start:
end:
NOP ;

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

```
SSYNC ;

/* Core reset - forces reboot */
RAISE 1 ;
```

Listing 17-2. System Reset in C Language

```
bfrom_SysControl(SYSCTRL_SYSRESET, 0, NULL);
```

Exiting Reset to User Mode

To exit reset while remaining in user mode, use the code shown in Listing 17-3.

Listing 17-3. Exiting Reset to User Mode

```
_reset:   P1.L = 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 */
...
```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and `EVTx` registers are initialized here.

Exiting Reset to Supervisor Mode

To exit reset while remaining in supervisor mode, use the code shown in Listing 17-4.

Listing 17-4. Exiting Reset by Staying in Supervisor Mode

```
_reset:
   P0.L = 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 */
...
```

Initcode (SDRAM Controller Setup)

Listing 17-5 shows an example of initcode to setup the SDRAM controller. The SDRAM controller must be initialized *before* data can be booted into it. Therefore, the SDRAM controller is typically initialized by an initcode or by the preboot functionality. The following initcode example assumes that the preboot did not do the job.

Listing 17-5. Example Initcode (SDRAM Controller Setup)

```
#include <defBF548.h>
.section initcode;
/*******SDRAM Setup************/
Setup_SDRAM:
/* save to stack following C conventions */
void initcode(ADI_BOOT_DATA* pBS)
{
  *pEBIU_RSTCTL |= DDRSRESET;
  *pEBIU_DDRCTL0 =
               SET_tRC(8)
|
               SET_tRAS(6)
|
               SET_tRP(2)
|
               SET_tRFC(10)
|
               SET_tREF(1041);
 *pEBIU_DDRCTL1 =
               SET_tWTR(2)
|
               DDR_DEVSIZE_512
|
               DDR_DEVWIDTH_16
|
               CS0
|
               DDR_DATAWIDTH
|
               SET_tWR(2)
|
               SET_tMRD(2)
```

```
|
                SET_tRCD(2);
```

```
 *pEBIU_DDRCTL2 =
              nREGE
|
              nDLLRESET
|
              CASLATENCY2
|
              BURSTLENGTH1|
              0;}
```

Initcode (Power Management Control)

The following example shows how to change PLL and the voltage regulator within an initcode. The example assumes that the preboot did not do the job already.

Listing 17-6. Changing PLL and Voltage Regulator

```
#include <blackfin.h>
void initcode (ADI_BOOT_DATA* pBS)
{
    ADI_SYSCTRL_VALUES mystruct;
    mystruct.uwVrCtl = 0x...;
    mystruct.uwPllCtl = 0x...;
    mystruct.uwPllDiv = 0x...;
    bfrom_SysControl(SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
                                SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
                                SYSCTRL_WRITE,
                                &mystruct, NULL);
}
```

Care must be taken that the reprogramming of the PLL does not break the communication with the booting host. For example, in the case of UART boot, the `UARTx_DLL` and `UARTx_DLH` registers must be updated to keep the old bit rate.

Initcode (NAND Flash Boot Mode Configuration)

Listing 17-7 shows an example of initcode to enable the advanced ECC options for NAND flash boot mode. The initcode is loaded while the NAND flash boot kernel is configured for the default boot mode. In this example, after the initcode sequence is executed, the NAND flash boot kernel is in Multiple Image Mode. This example also alters the number of address cycles for further accesses, which further optimizes the boot kernel for the attached NAND flash device.

Listing 17-7. Initcode Options with NAND Flash Boot Mode

```
#include <bfrom.h>
void initcode(ADI_BOOT_DATA* pBS)
{
    /* Create a pointer to the ADI_BOOT_NAND structure */
    ADI_BOOT_NAND *pNS;

    /* Set the pointer to ADI_BOOT_NAND */
    pNS = pBS->dUserLong;

    /* NAND Boot Kernel Configuration
       Mode:                                Multiple Image Mode
       Number of blocks between each image: 10
       Number of images:                    4
       Number of address cycles:            4
    */
    pNS->EccParity.uwBlockSkipFeature = 2;
    pNS->EccParity.uwBlockModifier = 10;
    pNS->EccParity.uwMaxCopies = 3;
    pNS->AddressCycles.uwNumAddressCycles = 4;
}
```

Quickboot With Restore From SDRAM

This example could be part of an advanced power saving concept. Assume the Blackfin is waking up from hibernate and processing any master boot mode. If the SDRAM has not been shut down, but was put in self-refresh mode, the content of the SDRAM will still be valid after wake up. The boot process would only have to initialize on-chip memories. Several boot blocks might be tagged by the `BFLAG_QUICKBOOT` flag.

Some applications might use a power-down handler that saves the contents of L1 memory to SDRAM before entering the hibernate state. Listing 17-8 assumes a suitable power-down handler was present that generated a partial boot stream in SDRAM at address 0x0001 0000 containing all the instructions required to restore the L1 memory contents.

Listing 17-8. Quickboot with Restore from SDRAM

```
void L1_recovery_initcode (ADI_BOOT_DATA *pBS)
{
      if (pBS->dFlags & BFLAG_WAKEUP) {
      bfrom_MemBoot((void*)0x00010000, BFLAG_RETURN, NULL);
      }
}
```

The boot stream generated at 0x0001 0000 will only be processed upon a wake-up condition. The `BFLAG_RETURN` ensures that the new instance of the boot kernel returns to the initcode rather than jumps to the `EVT1` vector.

XOR Checksum

Listing 17-9 illustrates how an initcode can be used to register a callback routine. The routine is called after each boot block that has the `BFLAG_CALLBACK` flag set. The calculated XOR checksum is compared against the block header `ARGUMENT` field. When the checksum fails, this example goes into idle mode. Otherwise control is returned to the boot kernel.

Since this callback example accesses the data after it is loaded, it would fail if the target address were in L1 instruction space. Therefore the `BFLAG_INDIRECT` flag should also be set. The `xor_callback` routine could then perform the checksum calculation at an intermediate storage place. The boot kernel transfers the data from the temporary buffer to the final destination after the callback routine returns.

In general, the block size is bigger than the size of the temporary buffer. Therefore, the boot kernel may need to divide the processing of a single block into multiple steps. The callback routine may also need to be invoked multiple times—every time the temporary buffer is filled up and once for the remaining bytes. The boot kernel passes the `dFlags` parameter, so that the callback routines knows whether it is called the first time, the last time or neither. The `dUserLong` variable in the `ADI_BOOT_DATA` structure is used to store the intermediate results between function calls.

Listing 17-9. XOR Checksum

```
s32 xor_callback(ADI_BOOT_DATA* pBS, ADI_BOOT_BUFFER* pCS, s32
dFlags)
{
      s32 i;
      if ((pCS!= NULL) && (pBS->pHeader!= NULL)) {
            if (dFlags & CBFLAG_FIRST) {
            pBS->dUserLong = 0;
            }
            for (i=0; i<pCS->dByteCount/sizeof(s32); i++)
{
            pBS->dUserLong^= ((s32 *)pCS->pSource)[i];
            }
            if (dFlags & CBFLAG_FINAL) {
            if (pBS->dUserLong!= pBS->pHeader->dArgument) {
            idle ();
            }
            }
      }
      return 0;
}
void xor_initcode (ADI_BOOT_DATA *pBS)
{
      pBS->pCallBackFunction = xor_callback;
}
```

Note that the callback routine is not volatile. It should not be overwritten by subsequent boot blocks. It can, however, be overwritten after processing the last block with `BFLAG_CALLBACK` flag set.

The checksum algorithm must be booted first and cannot protect itself. Problems can be avoided by letting initcode and callback execute directly from off-chip flash memory. The ADSP-BF54x processor processors provide a CRC32 checksum algorithm in the on-chip L1 instruction ROM, that can be used for booting under this scenario. For more information see "CRC Checksum Calculation" on page 17-48.

Direct Code Execution

This code example illustrates how to instruct the CCES or VisualDSP++ tools to generate a flash image that causes the boot kernel to start code execution at flash address 0x2000 0020 rather than performing a regular boot. See "Direct Code Execution" on page 17-37.

First, a 32-byte data block is defined in an assembly file that contains the initial block.

```
.section bootblock;
.global _firstblock;
.var _firstblock[4] = 0xAD7BD006,
0x20000020, 0x00000010, 0x00000010;
```

Then, the linker is instructed to map the initial block to address 0x2000 0000 in the LDF file.

```
MEMORY
{
      MEM_ASYNC0
      {
            START(0x20000000)
            END(0x23FFFFFF)
            TYPE(ROM)
      WIDTH(8)
}
}
PROCESSOR p0
{
```

```
        RESOLVE(_firstblock,0x20000000)
        RESOLVE(start,0x20000020)
        KEEP(start,_firstblock)
SECTIONS
{
        flash
        {
                INPUT_SECTION_ALIGN(4)
                INPUT_SECTIONS($OBJECTS(program)
$LIBRARIES(program))
                INPUT_SECTIONS($OBJECTS(bootblock))
} >MEM_ASYNC0
}
}
```

To invoke the elfloader utility, activate the meminit feature and use the command-line switches `-romsplitter` and `-maskaddr`. Refer to the application note *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)* for further details.

Managing PBS Pages in OTP Memory

The following code snips illustrate how to read and write OTP memory, as it is required for the Preboot Settings (PBS). For detailed description of OTP API functions `bfrom_OtpCommand()`, `bfrom_OtpRead()` and `bfrom_OtpWrite()` used here, see Chapter 4, “One-Time Programmable Memory”.

The first example reads PBS settings from OTP and stores them into an instance of the `ADI_PBS_BLOCK` structure. This is a union composite of the `ADI_PBS_HALFPAGES` or the `ADI_PBS_BITFIELDS` types. These structure types are defined in the `bfrom.h` header file. The `dPbsSet` variable describes the set of PBS pages of interest. A 0x00 value reads from OTP pages 0x18 to 0x1B. A 0x01 value reads from OTP pages 0x1C to 0x1F and so on.

Listing 17-10. Reading a Set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpRead(PBS00+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs00L));
bfrom_OtpRead(PBS00+dPbsSet*4,
OTP_UPPER_HALF,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs01L));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_UPPER_HALF,
&(PBS.HalfPages.uqPbs01H));
bfrom_OtpRead(PBS02+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs02L));
bfrom_OtpCommand(OTP_CLOSE, 0);
```

The next example shows how PBS pages can be written.

Listing 17-11. Programming a Set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
/* fill PBS with meaningful data */
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00L));
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01L));
```

```
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01H));
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02L));
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02H));
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03L));
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03H));
bfrom_OtpCommand(OTP_CLOSE, 0);
```

If a set of PBS pages has been written earlier, but need to be replaced by a new set, the old PBS pages have to be invalidated. Do not use the `OTP_CHECK_FOR_PREV_WRITE` option in this case.

Listing 17-12. Invalidating a Set of PBS Pages

```
#include <blackfin.h>
#include <bfrom_h>
u32 dPbsSet = 0;
u64 dlInvalidate = (u64)0xC000000000000000;
bfrom_OtpWrite(PBS00+dPbsSet*4,
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
OTP_LOWER_HALF | OTP_NO_ECC, &dlInvalidate);
bfrom_OtpCommand(OTP_CLOSE, 0);
dPbsSet++;
/* write next set as in Listing x-2 */
```

For production you may want to lock the PBS pages to protect them from being overwritten in the field. This can be performed by the following instructions:

Listing 17-13. Write-protecting a Set of PBS Pages

```
#include <blackfin.h>
#include <bfrom.h>
u32 dPbsSet = 0;
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpCommand(OTP_CLOSE, 0);
```

When locking PBS pages remember to duplicate the active set of PBS pages best reliability. In the above examples, if the `dPbsSet*4` contains the final configuration, then program set 5 with the same data. For completeness, note that the above code example does not lock the ECC fields corresponding to the PBS pages. See Chapter 4, “One-Time Programmable Memory” for details.

18 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the Blackfin processor and includes the following sections:

- "Phase-Locked Loop and Clock Control" on page 18-1
- "Dynamic Power Management Controller" on page 18-7
- "PLL and VR Registers" on page 18-25
- "System Control ROM Function" on page 18-29
- "Programming Examples" on page 18-35

Phase-Locked Loop and Clock Control

The input clock into the processor, CLKIN, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip, phase-locked loop (PLL) module. During normal operation, the user programs the PLL with a multiplication factor for CLKIN. The resulting, multiplied signal is the voltage controlled oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (CCLK).

Another user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the peripheral access bus (PAB), DMA access bus (DAB), external access bus (EAB), and the external bus interface unit (EBIU).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL divide register (PLL_DIV), select a divider value that allows these buses to run at or below the maximum SCLK rate specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to change dynamically in a "coarse adjustment." For a "fine adjustment," the PLL clock frequency can also be varied.

PLL Overview

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the dynamic power management controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see "Dynamic Power Management Controller" on page 18-7.

Subject to the maximum VCO frequency, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

Figure 18-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

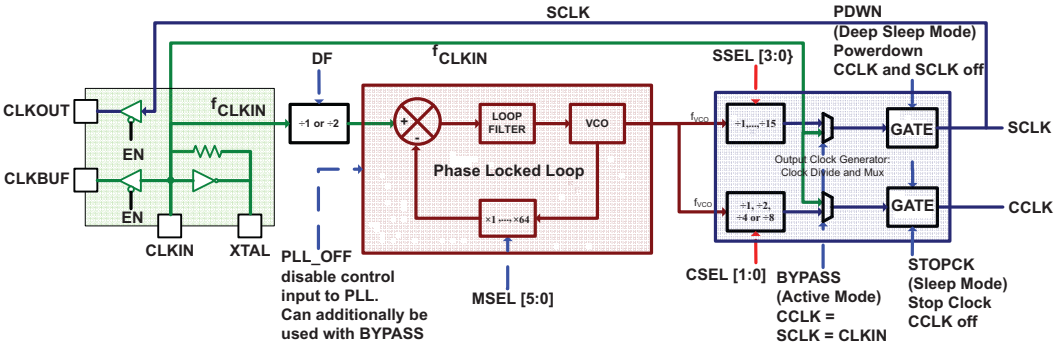


Figure 18-1. PLL Block Diagram

PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL Control (PLL_CTL) Register" on page 18-26.

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 0x8. This value can be reprogrammed at startup in the boot code.

Table 18-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 18-1. MSEL Encodings

Signal Name MSEL[5:0]	VCO Frequency DF = 0	DF = 1
0	64x	32x
1	1x	0.5x
2	2x	1x
N = 3–62	Nx	0.5Nx
63	63x	31.5x

The PLL control register (PLL_CTL) controls operation of the PLL (see Figure 18-5 on page 18-26). 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. This is handled by the System Control ROM Function (SysControl), shown on page 18-29.

Core Clock/System Clock Ratio Control

Table 18-2 describes the programmable relationship between the VCO frequency and the core clock. Table 18-3 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to

limit the SCLK to a frequency specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*. The SCLK drives all synchronous, system-level logic.

The divider ratio control bits, CSEL and SSEL, are in the PLL divide register (PLL_DIV). For information about this register, see "PLL Divide (PLL_DIV) Register" on page 18-26.

The reset value of CSEL[1:0] is 0x0, and the reset value of SSEL[3:0] is 0x4. These values can be reprogrammed at startup by the boot code.

By updating PLL_DIV with an appropriate value, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

Unlike writing the PLL_CTL register, the PLL_DIV register can be updated at any time to change the CCLK and SCLK divide values without the PLL programming sequence.

Table 18-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
00	1	300	300
01	2	600	300
10	4	600	150
11	8	400	50

As long as the MSEL and DF control bits in the PLL control register (PLL_CTL) remain constant, the PLL is locked.

Table 18-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	100	100
0010	2:1	200	100
0011	3:1	400	133
0100	4:1	500	125
0101	5:1	600	120
0110	6:1	600	100
N = 7–15	N:1	600	600/N

If changing the clock ratio through writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency. The PLL lock count register (PLL_LOCKCNT) defines the number of CLKIN 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 CLKIN cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see "Operating Modes" on page 18-7.

Dynamic Power Management Controller

The dynamic power management controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor's performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See "Operating Modes" on page 18-7.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.
- Voltage control – The processor provides an on-chip switching regulator controller which, with some external components, can generate internal voltage levels from the external V_{DD} (V_{DDEXT}) supply.

  Depending on the needs of the system, the voltage level can be reduced to save power. See "Controlling the Voltage Regulator" on page 18-17.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 18-4 summarizes the operational characteristics of each mode.

Table 18-4. Operational Characteristics

Operating Mode	Power Savings	PLL		CCLK	SCLK	Allowed DMA Access
		Status	Bypassed			
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The active and full on states of the DPMC/PLL can be determined by reading the PLL status register (see "PLL Status (PLL_STAT) Register" on page 18-27). In these modes, the core can either execute instructions or be in the idle core state. If the core is in the Idle state, it can be awakened by several sources.

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full On Mode

Full on mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full on mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. The system clock (`SCLK`) frequency is determined by the `SSEL`-specified ratio to `VCO`. DMA access is available to L1 and external memories. From full on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in Figure 18-2 on page 18-12.

Active Mode

In active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. DMA access is available to appropriately configured L1 and external memories.

In active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to full on or sleep modes.

From active mode, the processor can transition directly to full on, sleep, or deep sleep modes.

In this mode or in the transition phase to other modes, changes to MSEL are not latched by the PLL.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the core processor. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. Since CCLK is disabled, DMA access is available only to external memory in sleep mode. From sleep mode, a wake-up event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full on mode if the BYPASS bit is cleared

The processor resumes execution from the program counter value present immediately prior to entering sleep mode.

The STOPCK bit is not a status bit and is therefore unmodified by hardware when the wakeup occurs. Software must explicitly clear STOPCK in the next write to PLL_CTL to avoid going back into sleep mode.

Deep Sleep Mode

Deep sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals except the real-time clock (RTC) are disabled. DMA is not supported in this mode.

Deep sleep mode can be exited only by a hardware reset event or an RTC interrupt. A hardware reset begins the hardware reset sequence. An RTC interrupt causes the processor to transition to active mode, and execution resumes from where the program counter was when deep sleep mode was entered. If an interrupt is also enabled in SIC_IMASK0, the vector is taken immediately after exiting deep sleep and the ISR is executed.

Note an RTC interrupt in deep sleep mode automatically resets some fields of the PLL control register (PLL_CTL). See Table 18-5.

When in deep sleep mode, clocking to the DDR is turned off. Before entering deep sleep mode, software should ensure that important information in DDR memory is saved to a non-volatile memory and/or the DDR is placed into self-refresh mode.

Table 18-5. PLL_CTL Values After RTC Wake-up Interrupt

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 18-2. Since this feature is coupled to the on-chip switching regulator controller, it is discussed in detail in "Powering Down the Core (Hibernate State)" on page 18-20.

Operating Mode Transitions

Figure 18-2 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes and rectangles represent processor states. Arrows show the allowed transitions into and out of each mode or state.

For mode transitions, the text next to each transition arrow shows the fields in the PLL control register (`PLL_CTL`) that must be changed for the transition to occur. For example, the transition from full on mode to sleep mode indicates that the `STOPCK` bit must be set to 1 and the `PDWN` bit must be set to 0.

For transitions to processor states, the text next to each transition arrow shows either a processor event (for example, RTC wake-up or hardware reset) or the fields in the voltage regulator control register (`VR_CTL`) that must be changed for the transition to occur.

For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 18-14.

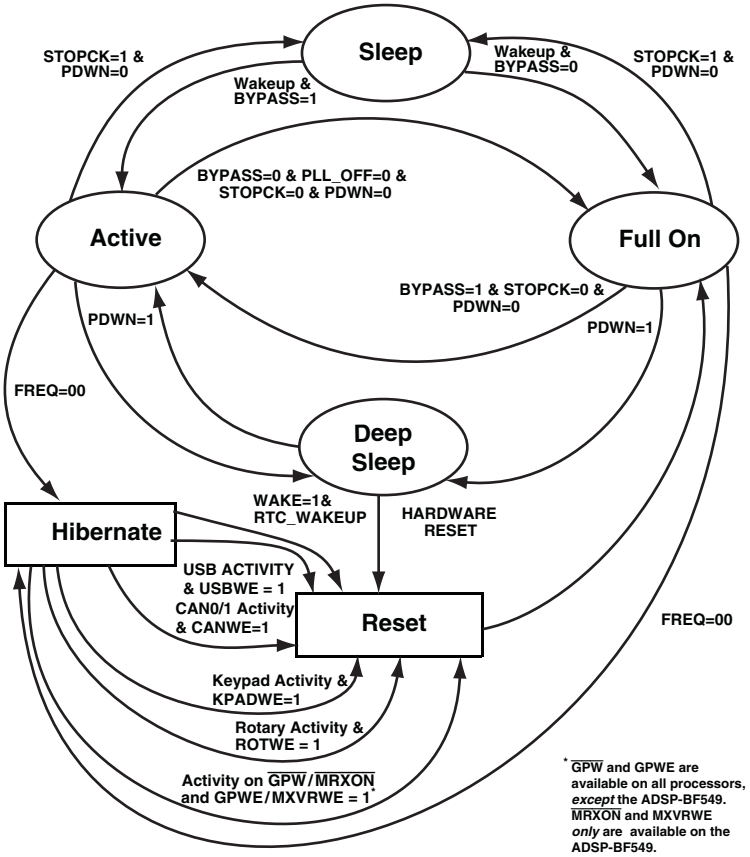


Figure 18-2. Operating Mode Transitions

In addition to the mode transitions shown in Figure 18-2, power to the PLL can be applied and removed while in the active operating mode.

Changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect:

- PLL disabled: In addition to being bypassed in the active mode, the PLL can be disabled.

  When the PLL is disabled, additional power savings are achieved although they are relatively small. To disable the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- PLL enabled: When the PLL is disabled, it can be re-enabled later when additional performance is required.

  The PLL must be re-enabled before transitioning to full on or sleep operating modes. To re-enable the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- New multiplier ratio: The clock-in to VCO clock (CLKIN to VCO) multiplier ratio can also be changed while in full on mode.

  The PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full on mode. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence.

Table 18-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 18-6 causes unpredictable behavior.

Table 18-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	–
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control register (PLL_CTL). Merely modifying the bits of the PLL_CTL register does not change the operating mode or the behavior of the PLL. Changes to the PLL_CTL register are realized only after executing a specific code sequence. This sequence is managed by an user-callable routine in the on-chip ROM called bfrom_SysControl(). When calling this function, no further precautions have to be taken.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or the changes reapply power to the PLL, the PLL needs to relock. To relock, the PLL lock counter is first cleared, and then it begins incrementing, once per SCLK cycle. After the PLL lock counter reaches the value programmed into the PLL lock count register (PLL_LOCKCNT), the PLL sets the PLL_LOCKED bit in the PLL status register (PLL_STAT), and the PLL asserts the PLL wake-up interrupt.

When the bfrom_SysControl() routine reprograms the PLL_CTL register with a new value, it executes a subsequent IDLE instruction. It prevents all other system interrupt sources other than the DPMC from waking the core up from the idle state. If the lock counter expires, the PLL issues an interrupt and the code execution continues with the instruction after the IDLE instruction. Therefore, the system is in the new state by the time the bfrom_SysControl() routine returns.

If the new value written to the PLL_CTL or VR_CTL register is the same as the previous value, the PLL wake-up occurs immediately (PLL is already locked), but the core and system clock are bypassed for the PLL_LOCKCNT duration. For this interval, code executes at the CLKIN rate instead of at the expected CCLK rate. Software guards against this condition by comparing the current value to the new value before writing the new value.

When the wake-up signal is asserted, the processor continues, causing a transition to:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full on mode if the BYPASS bit is cleared

If the PLL_CTL register is programmed to enter the sleep operating mode, the processor immediately transitions to the sleep mode and waits for a wake-up signal before continuing.

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 to return from the bfrom_SysControl() routine.
- A hardware reset causes the processor to execute the reset sequence. For more information about hardware reset, see Chapter 17, “System Reset and Booting”

If no operating mode transition is programmed, the PLL generates a wake-up signal, and bfrom_SysControl() routine returns.

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core processor at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses multiple power domains. Each power domain has a separate V_{DD} supply. Note that the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for details on the allowed voltage ranges for each power domain and power dissipation data.

Power Supply Management

The processor provides an on-chip switching regulator controller which, with some external hardware, can generate internal voltage levels from the external V_{DDEXT} supply with an external power transistor as shown in Figure 18-3. This voltage level can be reduced to save power, depending upon the needs of the system.

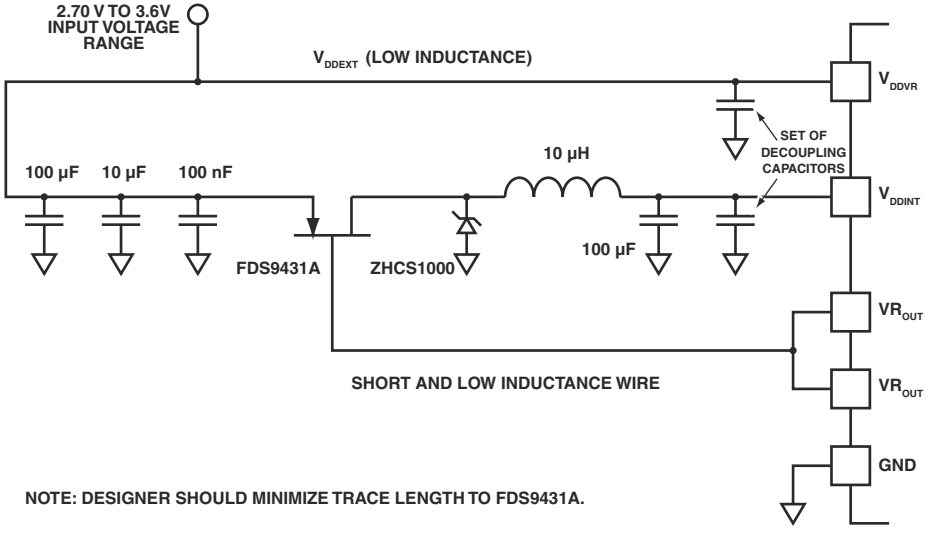


Figure 18-3. Processor Voltage Regulator

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 18-8 on page 18-28). The state of the `VR_CTL` register is maintained during power down modes and hibernate state. It is only set to its reset value by a powerup reset sequence. The `VR_CTL` register should not be written directly. Rather, the `bfrom_SysControl()` routine, which resides in the on-chip ROM, should be used to access it.

The on-chip switching regulator can be modified in terms of its transient behavior in the `GAIN` and `FREQ` fields of the `VR_CTL` register.

The two-bit GAIN field controls the internal loop gain of the switching regulator loop; this field controls how quickly the voltage output settles on its final value. In general, higher gain allows for quicker settling times but causes more overshoot in the process.

Table 18-7 lists the gain levels configured by GAIN[1:0].

Table 18-7. GAIN Encodings

GAIN	Value
b#00	5
b#01	10
b#10	20
b#11	50

The two-bit FREQ field controls the switching oscillator frequency for the voltage regulator. A higher frequency setting allows for smaller switching capacitor and inductor values, while potentially generating more EMI (electromagnetic interference).

Table 18-8 lists the switching frequency values configured by FREQ[1:0].

Table 18-8. FREQ Encodings

FREQ	Value
b#00	Powerdown/bypass onboard regulation
b#01	333 kHz
b#10	667 kHz
b#11	1 MHz

To bypass onboard regulation, program a value of b#00 in the FREQ field and leave the VROUT pins floating. Nevertheless, the VLEV field in the applied VR_CTL value should still reflect the applied voltage value.

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for more information about supported voltage levels, regulator tolerances, and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance requires significant changes to the operating voltage level. To ensure predictable behavior when varying the operating voltage, the processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. The four-bit voltage level (`VLEV`) field identifies the nominal internal voltage level. Refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the applicable `VLEV` voltage range and associated voltage tolerances.

Table 18-9 lists the voltage level values for `VLEV[3:0]`.

Table 18-9. VLEV Encodings

VLEV	Voltage
b#0000–b#0101	Reserved
b#0110	Reserved
b#0111	Reserved
b#1000	0.95 volts
b#1001	1.00 volts
b#1010	1.05 volts
b#1011	1.10 volts
b#1100	1.15 volts

Table 18-9. VLEV Encodings (Cont'd)

VLEV	Voltage
b#1101	1.20 volts
b#1110	1.25 volts
b#1111	1.30 volts

After changing the voltage level in the VR_CTL register, the PLL automatically enters the active mode when the processor enters the idle state. At that point, the voltage level changes and the PLL relocks with the new voltage. After the PLL_LOCKCNT has expired, the part returns to the full on state. When changing voltages, a larger PLL_LOCKCNT value may be necessary than when changing just the PLL frequency. See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for details.

After the voltage is changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the new operating voltage level.

Even if the internal voltage regulator is bypassed and the VDDINT voltage is applied by an external regulator, the bfrom_SysControl() routine must be called at startup or whenever the voltage changes at run time. Afterwards, the SYSCTRL_EXTVOLTAGE bit should be set along with a proper VLEV value in the VR_CTL register.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing b#00 to the FREQ bits of the VR_CTL register. This disables both CCLK and SCLK. Furthermore, it sets the internal power supply voltage (V_{DDINT}) to 0 V, eliminating any leakage currents from the processor. The internal supply regulator can be woken up by several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the $\overline{\texttt{RESET}}$ pin always exits hibernate state and requires no modification to VR_CTL.
- RTC event. Set the wake-up enable (WAKE) control bit to enable wake-up upon a RTC interrupt. This can be any of the RTC interrupts (alarm, daily alarm, day, hour, minute, second, or stopwatch).
- General-purpose event (all processors *except* ADSP-BF549). Set the general-purpose wake-up enable (GPWE) control bit to enable wake-up upon detection of an active low signal on the $\overline{\texttt{GPW}}$ pin.
- MXVR event (ADSP-BF549 processor *only*). Set the MXVR wake-up enable (MXVRWE) control bit to enable wake-up upon detection of an active low signal on the $\overline{\texttt{MRXON}}$ pin. For more information see Chapter 29, “Media Transceiver Module (MXVR)”.
- Activity on either CANxRX pin. Set the CAN RX wake-up enable (CANWE) control bit to enable wake-up upon detection of CAN bus activity on either of the CANxRX pins. For more information see Chapter 31, “CAN Module”.
- Activity on the rotary counter pins. Set the rotary counter wake-up enable (ROTWE) control bit to enable wake-up upon activity on the rotary counter pins. If any edge is detected on either the CUD or CDG pins, or if an active low state is detected on the CZM pin, this wake-up event is generated. For more information see Chapter 13, “Rotary Counter”.
- Activity on the keypad pins. Set the keypad wake-up enable (KPADWE) control bit to enable wake-up upon activity on the keypad pins. If an active low state is detected on any of the KEY_ROWx pins, this wake-up event is generated. For more information see Chapter 30, “Keypad Interface”.

- USB activity. Set the USB wake-up enable (USBWE) control bit to enable wake-up upon USB activity. If any edge is detected on the USB_DP, USB_DM, or USB_VBUS pins, this wake-up event is generated. For more information see Chapter 26, "USB OTG Controller".
- The hibernate functions will only work if V_{DDRTC} is supplied. This is the supply that is needed to maintain the VR_CTL register.

For the peripheral hibernate wake-up sources described above, a general-purpose wake-up can be implemented if the peripheral isn't used. For example, if MXVR is not used, an external host can be connected to the $\overline{\text{MRXON}}$ pin that holds the pin high until the wake-up is required. If MXVRWE is set, a transition to low on $\overline{\text{MRXON}}$ will exit hibernate state, and the host could be set up to provide this signal.

If the on-chip supply controller is bypassed so that V_{DDINT} is sourced externally, the only way to power down the core is to remove the external V_{DDINT} voltage source.

When the core is powered down, V_{DDINT} is set to 0 V, and so the internal state of the processor is not maintained, with the exception of the VR_CTL register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power. Be sure to set the SCKE-low-during-reset (SCKELOW) control bit in VR_CTL to protect against the default reset state behavior of setting the EBIU pins to their inactive state. Failure to set this bit results in the SCKE pin going high during reset, which takes the DDR out of self-refresh mode, resulting in data decay in the DDR due to loss of refresh rate.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a three-state level, unless otherwise specified.

To power down the internal supply:

1. Write 0 to the appropriate bits in the SIC_IWRx registers to prevent enabled peripheral resources from interrupting the hibernate process.
2. Call the bfrom_SysControl() routine, ensuring that the FREQ bits in the VR_CTL variable are set to b#00 and the appropriate wake-up bit(s) are set to 1 (USBWE, ROTWE, GPWE/MXVRWE, KPADWE, CANWE, WAKE). Optionally, set the SCKELOW bit if DDR data should be maintained.
3. The bfrom_SysControl() routine will execute until V_{DDINT} transitions to 0V. It never returns.
4. When the processor is woken up, the PLL relocks and the boot sequence defined by the BMODE[3:0] pin settings takes effect.

The WURESET in the SYSCTRL register is set and stays set until the next hardware reset. The WURESET bit may control conditional boot process.

If the CLKBUFOE bit is set, the crystal oscillator and CLKBUF signals remain enabled during hibernate and draw current.

Recovery From Hibernate State

When utilizing the hibernate state to maximize power savings, additional features of the ADSP-BF54x processor Blackfin processors can be used to coordinate system response and subsequent system activity when the processor resumes execution upon a hibernate wake-up event.

For the system outside of the Blackfin processor, the EXT_WAKE output pin is deasserted (driven low) when the Blackfin processor is about to enter the Hibernate state. This pin can be used in the system to signal an external component that it is now safe to remove the power supply. The state of the EXT_WAKE pin is not affected by the reset sequence, and no clock is

required for it to be driven. The EXT_WAKE pin is driven high when the on-chip regulator is again stable after resuming operation from the hibernate state.

For the Blackfin processor itself, the PLL_STAT register contains a set of wake-up status bits, which can be interrogated upon warm-boot to determine which source caused the wake-up event. This information can be useful to coordinate with external system components regarding lost traffic due to the previous activity causing a wake-up event rather than a processed message. For example, if a CAN message took the processor out of hibernate state, that message would not have been received by the processor because the processor would have had to perform a self-reset and boot and run the application before being able to actually handle a CAN message.

The SCKELOW bit in the VR_CTL register is maintained during the hibernate state. Typical use of this bit is to protect data in DDR memory during the hibernate state and subsequent reset event. Because of this, SCKELOW can be checked by software to determine whether the processor is being booted for the first time or if it is restarting after a hibernate event. If the application had set the bit prior to hibernate to protect the contents of DDR memory, the bit will read as 1 after the reset event takes place. This feature is useful if, for example, the desire is to shorten boot times as much as possible. For larger applications, anything resolved to external memory and preserved for the duration of the hibernate state does not need to boot again after the wake-up event takes place. Adding code to an initialization block that simply checks the SCKELOW bit provides the application with the ability to determine whether a full boot or some abridged boot is necessary to have the full application resolved to the internal and external memory spaces.

PLL and VR Registers

The user interface to the PLL and VR is through five memory-mapped registers (MMRs) shown in Table 18-10 and illustrated in Figure 18-4 through Figure 18-8.

Table 18-10. PLL/VR Register Mapping

Register Name	Description	Notes
PLL_CTL	"PLL Control (PLL_CTL) Register" on page 18-26	Requires reprogramming sequence when written
PLL_DIV	"PLL Divide (PLL_DIV) Register" on page 18-26	Can be written freely
PLL_STAT	"PLL Status (PLL_STAT) Register" on page 18-27	Monitors active modes of operation and wake-up events
PLL_LOCKCNT	"PLL Lock Count (PLL_LOCKCNT) Register" on page 18-27	Number of SCLKs allowed for PLL to relock
VR_CTL	"Voltage Regulator Control (VR_CTL) Register" on page 18-28	Requires PLL reprogramming sequence when written

All four 16-bit MMRs must be accessed with aligned 16-bit reads/writes.

PLL Divide (PLL_DIV) Register

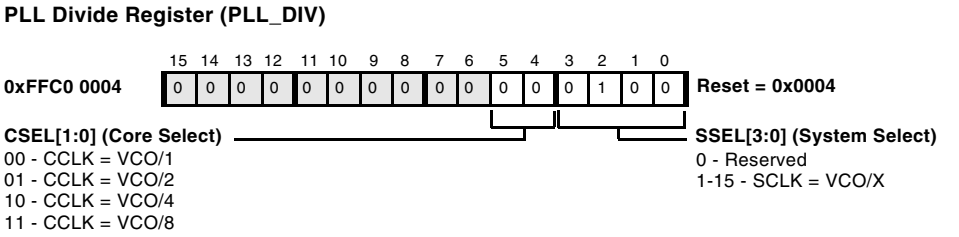


Figure 18-4. PLL Divide Register

PLL Control (PLL_CTL) Register

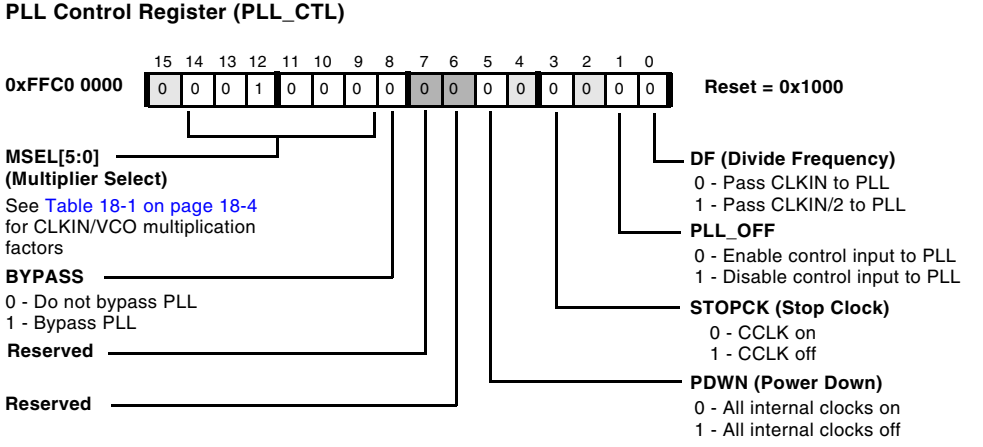


Figure 18-5. PLL Control Register

PLL Status (PLL_STAT) Register

PLL Status Register (PLL_STAT)

Read only. Unless otherwise noted, 1 - Processor operating in this mode. For more information, see "Operating Modes" on page 18-7.

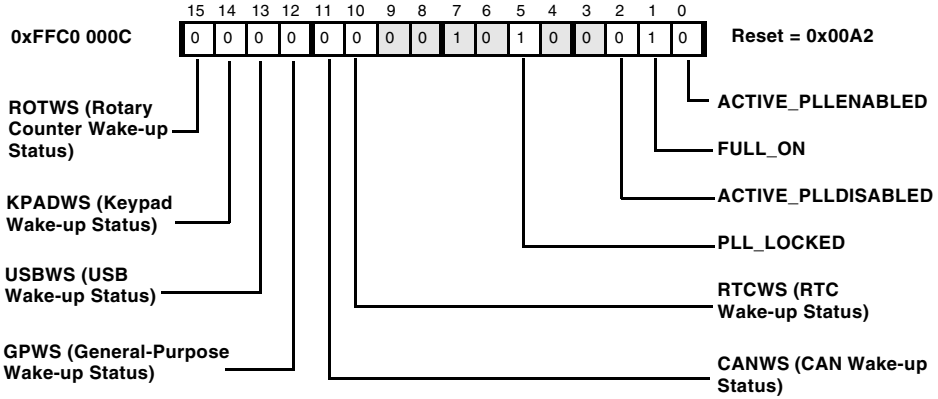


Figure 18-6. PLL Status Register

PLL Lock Count (PLL_LOCKCNT) Register

PLL Lock Count Register (PLL_LOCKCNT)

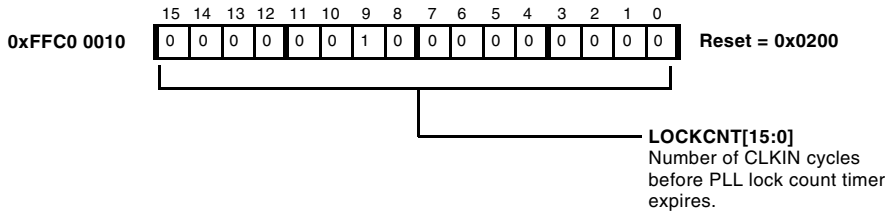


Figure 18-7. PLL Lock Count Register

Voltage Regulator Control (VR_CTL) Register

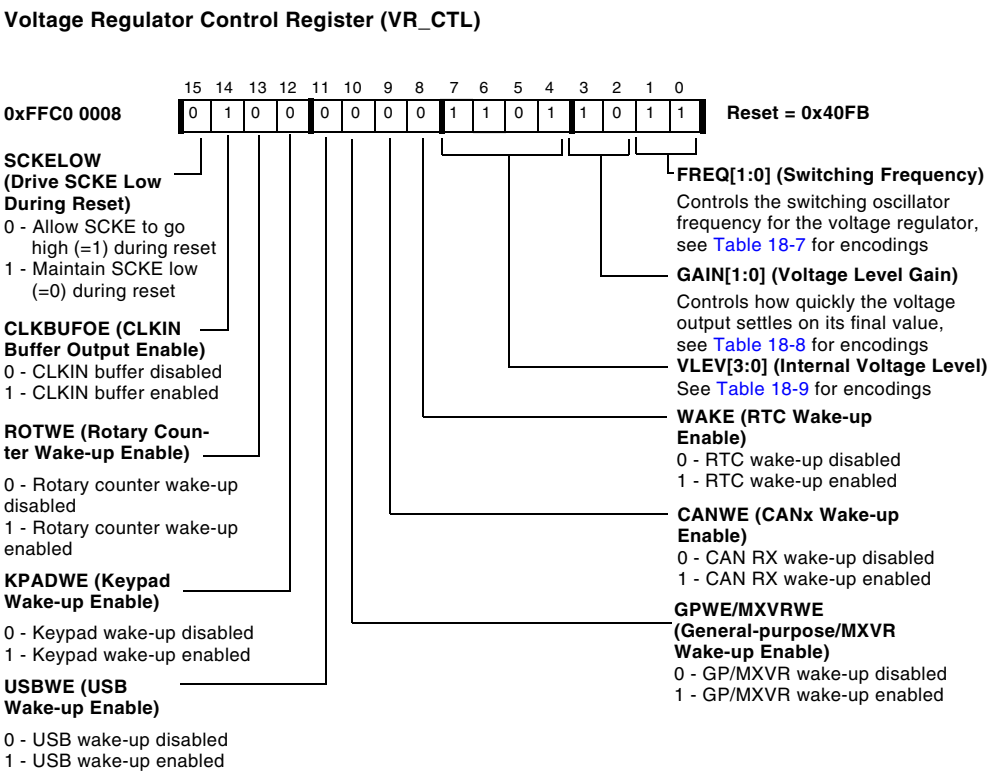


Figure 18-8. Voltage Regulator Control Register

Bit 10 enables the general-purpose wake-up (GPWE) for all processors *except* the ADSP-BF549. On ADSP-BF549 processors *only*, bit 10 enables the MXVR wake-up (MXVRWE).

The CLKIN buffer output enable (CLKBUFOE) control bit allows the Blackfin processor and another device to run from a single crystal oscillator. Clearing this bit prevents the CLKBUF pin from driving a buffered version of the input clock CLKIN.

System Control ROM Function

The PLL and voltage regulator registers should never be accessed directly. Rather, always use to the `bfrom_SysControl()` function to alter or read the register values. This function resides in the on-chip ROM and can be called by the user following C language style calling conventions.

Entry address: 0xEF00 0038

Arguments:

- `dActionFlags` word in `R0`
- `pSysCtrlSettings` pointer in `R1`
- `zero value` in `R2`

A potential error message from the internally called `bfrom_OtpRead()` function is forwarded and returned in `R0`.

The System Control ROM Function does not verify the correctness of the forwarded arguments. Therefore, it is up to the programmer to choose the correct values.

C prototype: `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);`

The first argument (`u32 dActionFlags`) holds the instruction flags. The following flags are supported:

```
#define SYSCTRL_READ        0x00000000
#define SYSCTRL_WRITE       0x00000001
#define SYSCTRL_SYSRESET    0x00000002
#define SYSCTRL_SOFTRESET   0x00000004
#define SYSCTRL_VRCTL       0x00000010
#define SYSCTRL_EXTVOLTAGE  0x00000020
#define SYSCTRL_INTVOLTAGE  0x00000000
#define SYSCTRL_OTPVOLTAGE  0x00000040
```

```
#define SYSCTRL_PLLCTL      0x00000100
#define SYSCTRL_PLLDIV      0x00000200
#define SYSCTRL_LOCKCNT     0x00000400
#define SYSCTRL_PLLSTAT     0x00000800
```

With SYSCTRL_READ and SYSCTRL_WRITE, a read or a write operation is initialized. SYSCTRL_SYSRESET performs a system reset, and SYSCTRL_SOFTRESET combines a core and a system reset. The SYSCTRL_EXTVOLTAGE and SYSCTRL_INTVOLTAGE indicate whether if VDDINT is supplied externally or generated by the on-chip regulator; SYSCTRL_OTPVOLTAGE is for factory purposes only. The last five flags (_VRCTL, _PLLCTL, _PLLDIV, _LOCKCNT, _PLLSTAT) tell the system control ROM function, which register to write or read. Remember, SYSCTRL_PLLSTAT is read only.

The second argument (ADI_SYSCTRL_VALUES *pSysCtrlSettings) passes a pointer to a special structure, which has entries for all PLL and voltage regulator registers. It is predefined in the bfrom.h header file as

```
typedef struct {

    u16 uwVrCtl;

    u16 uwPllCtl;

    u16 uwPllDiv;

    u16 uwPllLockCnt;

    u16 uwPllStat;

} ADI_SYSCTRL_VALUES;
```

The third argument is reserved and should always be kept zero (NULL pointer).

For the return value, see the description of the `bfrom_OtpRead()` ROM routine, whereby single-bit warnings are suppressed.

The System Control ROM Function automatically performs the correct programming sequence for the Dynamic Power Management System of the Blackfin processor.

Programming Model

The programming model for the Access System Control ROM Function in C/C++ and Assembly is described in the following sections.

Access System Control ROM Function in C/C++

To read the `PLL_DIV` and `PLL_CTL` register values, specify the `SYSCTRL_READ` instruction flag along with `SYSCTRL_PLLCTL and SYSCTRL_PLLDIV` register flags. The `bfrom_OtpRead()` function then only updates the `uwPllCtl` and `uwPllDiv` variables.

```
ADI_SYSCTRL_VALUES read;
bfrom_SysControl ( SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_READ, &read, NULL );
```

The values of the `PLL_CTL` and `PLL_DIV` registers can be accessed in the `read.uwPllCtl` and `read.uwPllDiv`, respectively.

To update register values, specify the `SYSCTRL_WRITE` instruction flag along with the register flags of those register that should be modified and have valid data in the respective `ADI_SYSCTRL_VALUES` variables.

```
ADI_SYSCTRL_VALUES write;
write.uwPllCtl = 0x1400;
write.uwPllDiv = 0x0005;
bfrom_SysControl ( SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_WRITE, &write, NULL );
```

Access System Control ROM Function in Assembly

The assembler supports C structs. Here it is required to import the file `bfrom.h`.

```
#include <bfrom.h>
.IMPORT “bfrom.h”;
.STRUCT ADI_SYSCTRL_VALUES dpm;
```

You are free to pre-load the struct:

```
.STRUCT ADI_SYSCTRL_VALUES dpm = { 0x40DB, 0x1400, 0x0005,
0x0200, 0x00A2 };
```

You can also load the values dynamically inside the code:

```
P5.H = hi(dpm);

P5.L = lo(dpm->uwVrCtl);
R7 = 0x40DB (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllCtl);
R7 = 0x1400 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllDiv);
R7 = 0x0005 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllLockCnt);
R7 = 0x0200 (z);
w[P5] = R0;
```

The function `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);` can be accessed by `bfrom_syscontrol`. Following the C/C++ run-time environment conventions, the parameters passed are held by the data registers `R0`, `R1` and `R2`.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Always allocate at least 12 bytes on the stack for outgoing
arguments, even if the function being called requires less than
this. */
SP += -12;

R0 = ( SYSCTRL_VRCTL      |
       SYSCTRL_INTVOLTAGE |
       SYSCTRL_PLLCTL     |
       SYSCTRL_PLLDIV     |
       SYSCTRL_WRITE      );
R1.H = hi(dpm);
R1.L = lo(dpm);
R2 = 0 (z);
P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

As an alternative for taking a C-struct, the processor's internal scratchpad memory can be used too. Therefore, the stack/frame pointer must be loaded and passed.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Always allocate at least 12 bytes on the stack for outgoing
arguments, even if the function being called requires less than
this. */
SP += -12;

R7 = 0;
R7.L = 0x40DB;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R7;
R7.L = 0x1400;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R7;
R7.L = 0x0005;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R7;
R7.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R7;

R0 = ( SYSCTRL_VRCTL      |
       SYSCTRL_INTVOLTAGE |
       SYSCTRL_PLLCTL     |
       SYSCTRL_PLLDIV     |
       SYSCTRL_WRITE      );
```

```
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0;

P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

Programming Examples

The following code examples illustrate how to use the System Control ROM Function in running various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- PLL control (`PLL_CTL`) register setting: 0x1400
- PLL divider (`PLL_DIV`) register setting: 0x0005
- PLL lock count (`PLL_LOCKCNT`) register setting: 0x0200
- Clock in (`CLKIN`) frequency: 25MHz

VCO frequency is 250MHz, core clock frequency is 250MHz and system clock frequency is 50MHz.

- Voltage regulator control (`VR_CTL`) register setting: 0x40DB
- Logical voltage level (`VDDINT`) is at 1.20V

For operating mode transition and voltage regulator examples:

- C

```
#include <blackfin.h>

#include <bfrom.h>
```

- Assembly

```
#include <blackfin.h>

#include <bfrom.h>

.IMPORT “bfrom.h”;

#define IMM32(reg,val) reg##.H=hi(val); reg##.L=lo(val)
```

Full On Mode to Active Mode and Back

Listing 18-1 and Listing 18-2 provide code for transitioning from full on operating mode to active mode, in C and Blackfin assembly code, respectively.

Listing 18-1. Transitioning from Full On Mode to Active Mode (C)

```
void active(void)
{
ADI_SYSCTRL_VALUES active;
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &active, NULL );
active.uwPllCtl |= (BYPASS | PLL_OFF); /* PLL_OFF bit optional */
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_WRITE, &active, NULL );
return;
}
```

Listing 18-2. Transitioning from Full On Mode to Active Mode (ASM)

```
__active:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_READ );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(BYPASS));
bitset(R0,bitpos(PLL_OFF));
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__active.end:
```

To return from active mode (go back to full on mode), the BYPASS bit and the PLL_OFF bit, respectively, must be cleared again.

Transition to Sleep Mode or Deep Sleep Mode

Listing 18-3 and Listing 18-4 provide code for transitioning from full on operating mode to sleep or deep sleep mode, in C and Blackfin assembly code, respectively.

Listing 18-3. Transitioning to Sleep Mode or Deep Sleep Mode, respectively (C)

```
void sleep(void)
{
ADI_SYSCTRL_VALUES sleep;
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &sleep, NULL );
active.uwPllCtl |= STOPCK; /* either: Sleep Mode */
active.uwPllCtl |= PDWN; /* or: Deep Sleep Mode */
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_WRITE, &sleep, NULL );
return;
}
```

Listing 18-4. Transitioning to Sleep Mode or Deep Sleep Mode, respectively (ASM)

```
__sleep:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_READ );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(STOPCK)); /* either: Sleep Mode */
bitset(R0,bitpos(PDWN)); /* or: Deep Sleep Mode */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__sleep.end:
```

Setting Wakeups and Entering Hibernate State

Listing 18-5 and Listing 18-6 provide code for configuring the regulator wakeups (RTC wakeup) and placing the regulator in the hibernate state, in C and Blackfin assembly code, respectively.

Listing 18-5. Configuring Regulator Wakeups and Entering Hibernate State (C)

```
void hibernate(void)
{
ADI_SYSCTRL_VALUES hibernate;
/* SCLKELOW = 1: Enable Drive CKE Low During Reset */
/* Protect DDR contents during reset after wakeup */
hibernate.uwVrCtl = SCKELOW |
        WAKE     | /* RTC/Reset Wake-Up Enable */
        nCANWE    | /* CAN Wake-Up Disable */
        nGPWE     | /*General-Purpose Wake-Up Disable/*
        nUSBWE    | /* USB Wake-Up Disable */
        nKPADWE   | /* Keypad Wake-Up Disable */
        nROTWE    | /* Rotary Wake-Up Disable */
        nCLKBUFOE | /* CLKIN Buffer Output Disable */
        HIBERNATE; /* Powerdown/Bypass On-Board Regulation */
bfrom_SysControl( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
SYSCTRL_WRITE, &hibernate, NULL );
/* Hibernate State: no code executes until wakeup triggers reset
*/
}
```

Listing 18-6. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```
__hibernate:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

cli R6; /* disable interrupts, copy IMASK to R6 */

/* SCLKELOW = 1: Enable Drive CKE Low During Reset */
/* Protect DDR contents during reset after wakeup */
R0.L = SCKELOW |
        WAKE      | /* RTC/Reset Wake-Up Enable */
       nCANWE     | /* CAN Wake-Up Disable */
       nGPWE      | /* General-Purpose Wake-Up Disable */
       nUSBWE     | /* USB Wake-Up Disable */
       nKPADWE    | /* Keypad Wake-Up Disable */
       nROTWE     | /* Rotary Wake-Up Disable */
       nCLKBUFOE  | /* CLKIN Buffer Output Disable */
        HIBERNATE ; /* Powerdown/Bypass On-Board Regulation */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;

R0 = ( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
/* Hibernate State: no code executes until wakeup triggers reset
*/

__hibernate.end:
```

Perform a System Reset or Soft-Reset

Listing 18-7 and Listing 18-8 provide code for executing a system reset or a soft-reset (= system reset + core reset), in Blackfin assembly and C code, respectively.

Listing 18-7. Execute a System Reset or a Soft-Reset

```
void reset(void)
{
bfrom_SysControl( SYSCTRL_SYSRESET, NULL, NULL ); /* either */
bfrom_SysControl( SYSCTRL_SOFTRESET, NULL, NULL ); /* or */
return;
}
```

Listing 18-8. Listing 8. Execute a System Reset or a Soft-Reset

```
__reset:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = ( SYSCTRL_SYSRESET ); /* either */
R0 = ( SYSCTRL_SOFTRESET ); /* or */
R1 = 0 (z);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
__reset.end:
```

Change VCO, Core Clock, and System Clock Frequency

Listing 18-9 and Listing 18-10 provide code for changing the CLKIN to VCO multiplier (from 10x to 21x), keeping CSEL divider at (1) and changing the SSEL divider (from 5 to 4) in full on operating mode, in C and Blackfin assembly code, respectively.

Listing 18-9. Transition of Frequencies (C)

```
void frequency(void)
{
ADI_SYSCTRL_VALUES frequency;

/* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
frequency.uwPllCtl = SET_MSEL(21) ;

/* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
/* CCLK = VCO / 1 */
frequency.uwPllDiv = SET_SSEL(4) |
CSEL_DIV1 ;

frequency.uwPllLockCnt = 0x0200;
```

```
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT | SYSCTRL_WRITE, &frequency,
NULL );
return;
}
```

Listing 18-10. Transition of Frequencies (ASM)

```
__frequency:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

/* write the struct */
R0 = 0;

R0.L = SET_MSEL(21) ; /* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0.L = SET_SSEL(4) | /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
       CSEL_DIV1   ; /* CCLK = VCO / 1 */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;

R0.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;

/* argument 1 in R0 */
R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_WRITE );
```

```
/* argument 2 in R1: structure lays on local stack */
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);

/* argument 3 must always be NULL */
R2 = 0;

/* call of SysControl function */
IMM32(P4,BFROM_SYSCONTROL);
call (P4); /* R0 contains the result from SysControl */

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__frequency.end:
```

Changing Voltage Levels

Listing 18-11 and Listing 18-12 provide code for changing the voltage level dynamically, in C and Blackfin assembly code, respectively. The voltage level will be changed to 1.25V. Additional code may be required to alter the core clock frequency when voltage level is being decreased. Refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the applicable `VLEV` voltage range and associated supported core clock speeds.

Listing 18-11. Changing Core Voltage via the On-Chip Regulator (C)

```
void voltage(void)
{
ADI_SYSCTRL_VALUES voltage;
voltage.uwVrCtl = VLEV_125  | /* VLEV = 1.25 V */
                  CLKBUFOE  | /* CLKIN Buffer Output Enable */
                  GAIN_20   | /* GAIN = 20 */
                  FREQ_1000 ; /* Switching Frequency Is 1 MHz */
bfrom_SysControl( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
SYSCTRL_WRITE, &voltage, NULL );
return;
}
```

Listing 18-12. Changing Core Voltage through the On-Chip Regulator (ASM)

```
__voltage:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0.L = VLEV_125  | /* VLEV = 1.25 V */
       CLKBUFOE  | /* CLKIN Buffer Output Enable */
       GAIN_20   | /* GAIN = 20 */
       FREQ_1000 ; /* Switching Frequency Is 1 MHz */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof (ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
```

```
R0 = ( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__voltage.end:
```

The previous sequence must also be executed when the V_{DDINT} voltage is applied externally to ensure internal timings can appropriately be adjusted for the constant or changing V_{DDINT} voltage. In this case, replace the `SYSCTRL_INTVOLTAGE` flag with the `SYSCTRL_EXTVOLTAGE` flag.

19 SYSTEM DESIGN

This chapter provides hardware, software, and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, the design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference is made to the corresponding section of the text, instead of repeating the discussion in this chapter.

- "Pin Descriptions" on page 19-1
- "Managing Clocks" on page 19-2
- "Configuring and Servicing Interrupts" on page 19-2
- "Semaphores" on page 19-3
- "Data Delays, Latencies, and Throughput" on page 19-4
- "Bus Priorities" on page 19-5
- "System-Level Hardware Design" on page 19-5
- "Recommended Reading" on page 19-19

Pin Descriptions

Refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for pin information, including pin numbers for the 400-ball MBGA.

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's CLKIN pin. It is not possible to halt, change, or operate CLKIN below the specified frequency during normal operation. The processor uses the clock input (CLKIN) to generate on-chip clocks. These include the core clock (CCLK) and the peripheral clock (SCLK).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed through the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed through the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see "Dynamic Power Management" on page 18-1.

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped through the system interrupt assignment registers (SIC_IARx). For more information, see Chapter 6, "System Interrupts".

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished through semaphores.

Semaphore coherency is guaranteed by using the test and set byte (atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 19-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 19-1. Query Semaphore

```
/* Query semaphore. Denotes "Busy" if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore is granted to current thread, and all
other contending threads are postponed because semaphore value at
[P0] is nonzero. Current thread could write thread_id to sema-
phore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an
operating system call to reschedule the current thread. */
```

Data Delays, Latencies, and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to Chapter 2, "Chip Bus Hierarchy".

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to Chapter 2, "Chip Bus Hierarchy".

System-Level Hardware Design

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

External Memory Design Issues

This section describes design issues related to external memory.

DDR Memory

The DDR controller has a dedicated set of pins that require special attention in board design and layout:

- Follow the recommendations of the DDR memory manufacturer. A good example of DDR layout recommendations is: *TN-46-14: Hardware Tips for Point-to-Point System Design* from Micron Technology.
- Proper board design and trace length matching is a critical part of reducing the DQS to DQ and DQM skew of any DDR design. Proper clock and V_{REF} layout are also critical.

- When matching trace lengths in board layout, be careful of the shape of the serpentine pattern. Capacitive coupling between segments of the trace reduces the effectiveness of length matching. About 20 mils is a good distance between segments of the same trace.
- Use serial termination on all data, address and control signals (except the CLKs) for small memory systems of one to four chips. Four or more devices may require parallel termination to a separately-generated V_{REF} instead of the serial termination.
- The maximum trace length of DQS, DQM and DQ signals should be less than 3.5 inches. This is a maximum total length for each signal measured by adding the length before, after and through any serial termination.
- Signal DDR_VSSR on the Blackfin processor is a shield signal to reduce noise on the DDR_VREF signal. It should be connected to ground right at the ball.
- DDR_VREF is a standard DDR signal with a voltage value of VDD_DDR/2. Place a DDR_VREF filtration capacitor to ground within 0.1 inch of the ball on the Blackfin processor. The signal should be derived in the standard way using 1% resistors and 0.1uF capacitor or capacitors even if using mobile DDR devices that do not also need the DDR_REF signal.
- The ADSP-BF54x processor processor is available with either a DDR SDRAM or a Mobile DDR SDRAM controller module on-chip. Each of these has different specifications. See *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for the proper nominal voltage and working voltage range.
- DDR and Mobile DDR timing calculations can result in margins that are measured in picoseconds. Therefore, they are complicated by the characteristics of the printed circuit board. To calculate, setup, hold, and skew values, consider contributions from the

printed circuit board, the controller and the memory device. Printed circuit board timing values can be difficult to determine and can change during the manufacturing process. Board simulation using IBIS models is useful. Be aware that reference plane noise and crosstalk can change slew rates and shift the time of arrival of signal edges in ways that may not be predicted by simulations.

- Mobile DDR, 166MHz or faster speed grade memory devices will give the setup, hold and skew margins required for PCB parameters. 133MHz devices will have zero PCB margin. 200MHz Mobile memory will give even larger tolerance for printed circuit board designs. Special attention should be paid to the recommendations for reduced trace length, matched trace length, and VREF trace width and filtering. Additionally, cross talk and noise can be reduced with careful use of stripline traces and quiet reference planes.

Memory Bus Pin Muxing and Flow Control

DDR memory has a complete set of dedicated functional pins and has no flow control.

All other parallel peripherals and memory types (such as SRAM, FLASH, BURST NOR FLASH, NAND FLASH) have dedicated pins and some pins that can be used for other functions such as GPIO.

Address bits 4 to 25 are muxed with GPIO functions on port I and port H and should be selected as address pins before using the asynchronous memory bus. Only the address pins used in the application need to be allocated as address pins. If high-order address bits are not needed, those pins may be used as GPIO. Note however that if booting from a parallel memory source, all 25 address pins and the BG and BGH pins are driven as outputs during boot time.

When using BURST NOR FLASH, select PI15/A25/NR_CLK for the NOR_CLK muxed function.

When using NAND, select PJ2/ND_RB and PJ1/ND_CE for the ND_RB and ND_CE muxed functions.

The bus request flow control pin is also muxed with GPIO functions. To use the asynchronous memory port, PJ11/AMC_BR should be selected for the AMC_BR function. This pin must then be held high with logic or a pullup resistor to allow bus transactions to be initiated by the processor.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 19-1 shows the interface to 8-bit SRAM or FLASH. Figure 19-2 shows the interface to 16-bit SRAM or FLASH

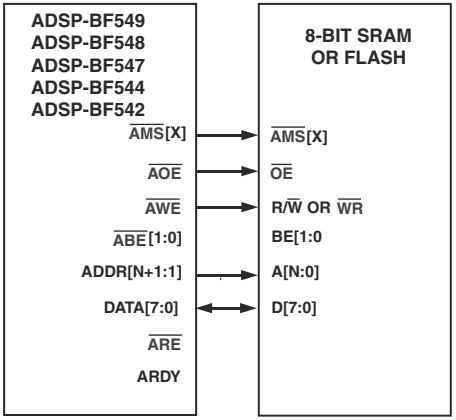


Figure 19-1. Interface to 8-Bit SRAM or FLASH

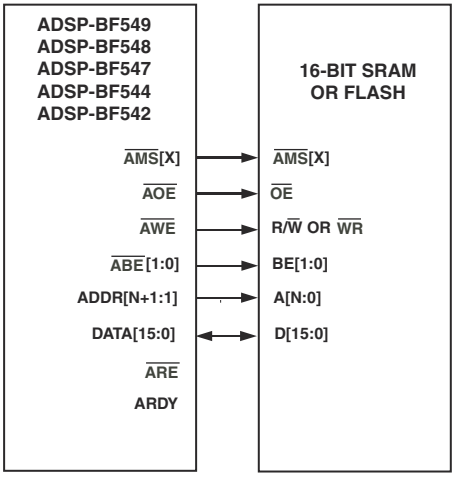


Figure 19-2. Interface to 16-Bit SRAM or FLASH

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend. Bus contention can also occur during reset or hibernation. This can be avoided by external resistors to inactivate chip selects.

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

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

BURST FLASH

The use of BURST FLASH is similar to that of asynchronous memory but requires special attention.

The burst flash connection requires only three special connections in addition to standard asynchronous memory signals (See Figure 19-3). The burst clock signal to the flash is provided on `PI15` and is the same pin as `A25`. The second signal requiring special attention is ADV provided by the Blackfin processor `AOE` pin. The `WAIT` output of the burst flash that should be connected directly to `ARDY`.

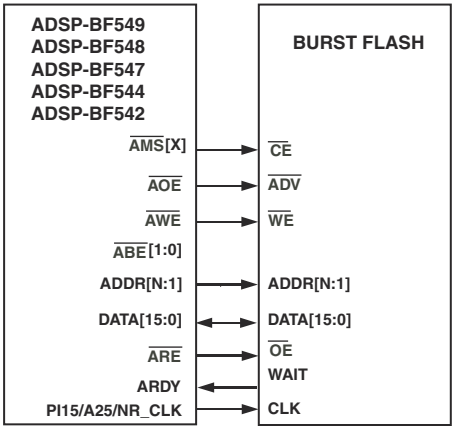


Figure 19-3. Interface to BURST FLASH

NAND FLASH

NAND FLASH shares the Asynchronous data bus. The NAND FLASH connection has many unique connections (See Figure 19-4). The chip enable is provided by PJ1/ND_CE while the ready busy signal is PJ2/ND_RB. PJ1/ND_CE requires a pull-up resistor because the general-purpose pins are inputs at power-up. Other unique signal functions include ND_CLE provided by ABE0 and ND_ALE provided by ABE1. I/O 0 to 7 or I/O 0 to 15 are supplied EBIU data pins D0 to D15.

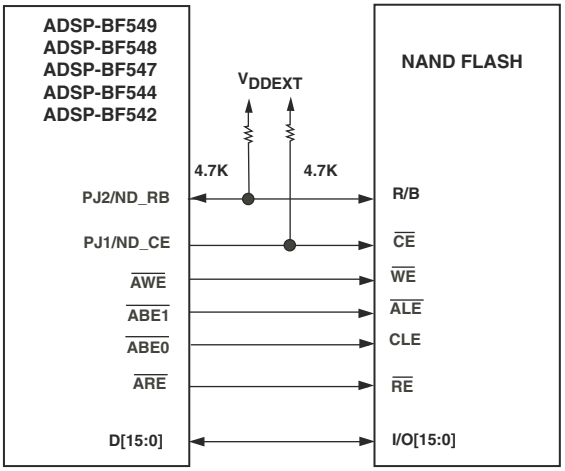


Figure 19-4. Interface to NAND FLASH

USB Controller

The UTMI (universal transceiver macro interface) of the USB controller is unique. It is what some companies call the PHY section of the USB controller. It has many features that allow connection directly to a USB cable connector. The important system hardware requirements are:

- The UTMI section of the USB does not use the system clock. An external clock is needed. The frequency must have an exact multiple to 480 MHz. The default value would be a 24 MHz clock to the USB_XI pin or a 24 MHz crystal circuit used with USB_XI and USB_XO. If using a crystal, use the same circuit as shown in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for CLKIN and XTAL.
- The UTMI section of the USB has our standard level of ESD protection. External protection diodes should be added at the connector to DP, DM, ID, and VBUS for proper ESD protection. There are several sources of ESD protection designed specifically for USB2.
- When operating in USB host mode, the user must supply an external 5 V supply at 8 ma or more. The 5 volts can be provided with a "charge pump" or a normal voltage regulator depending on the available input voltages available for the application. In either case recovery from VBUS error conditions require that it be enabled and disabled in software using a GPIO. It should have a resistor to set the initial value to disable the external 5 V source. The source should comply with On-The-Go Supplement to the USB 2.0 Specification Revision 1.0A. GPIO pin PE7 is used to enable this regulator in our software examples.
- DP and DM are intended for direct connection to the D+ and D- of a USB cable connector. They do not require any pull-up or pull-down resistors as these are applied internally by the UTMI in accordance with the programmed application mode. Note also that like any USB design, DP and DM should be routed as a differential pair with 90 to 100 Ohms mutual impedance.

- If using the USB in device mode only, you may put a pull-up resistor on the USB_ID pin or leave the pin disconnected. Either a pull-up resistor or leaving the pin disconnected will indicate device mode. If using the USB in host mode only, connect the USB_ID pin directly to ground.
- The USB_RST pin should be connected to an unpopulated resistor as there may be a future advantage to this configuration.
- The USB_VREF pin should be connected to a 0.1 mF capacitor to ground.
- As stated in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*, the 5 V tolerance of the UTMI pins is only true if V_{DDUSB} has some level of power. Some applications may anticipate V_{BUS} power from a device (perhaps located at the other end of the cable) when the local power is off. If this condition is expected to last for long periods measured in years, precautions should be taken to prevent long term damage to the product. One method for correcting this situation is to use the V_{BUS} power from the external device to power the VDDUSB pins of the processor.

ATAPI Bus

Special care is needed for ATAPI connections that require 5 V logic. Active voltage level translation buffers are required for any peripheral that uses 5 V logic levels.

Voltage Regulator

An internal voltage regulator can be used with the recommended external circuit to provide a flexible system of power management. Many applications require a fixed internal voltage value and can use a simple external voltage regulator to generate the V_{DDINT} supply voltage. The EXT_WAKE signal is provided to turn off the external voltage regulator when using the

hibernate operating mode. Because it is a high true power-up signal, it may be connected directly to the low true shutdown input of many common regulators.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Use simple signal integrity methods to prevent transmission line reflections that may cause extraneous extra clock and sync signals. Additionally, avoid overshoot and undershoot that can cause long term damage to input pins.

Some signals are especially critical for short trace length and usually require series termination. The CLKIN pin should have impedance-matching-series resistance at its driver. SPORT interface signals TCLK, RCLK, RFS, and TFS should use some termination. Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers often require resistive termination located at the source. (Note also that TFS and RFS should not be shorted in multichannel mode.) On the PPI interface, the PPI_CLK and SYNC signals also benefit from these standard signal integrity techniques. If these pins have multiple sources, it is difficult to keep the traces short. Consider termination of SDRAM clocks, control, address, and data to improve signal quality and reduce unwanted EMI.

Adding termination to fix a problem on an existing board requires delays for new artwork and new boards. A transmission line simulator is recommended for critical signals. IBIS models are available from Analog Devices Inc. that will assist signal simulation software. Some signals can be corrected with a small zero or 22 Ohm resistor located near the driver. The resistor value can be adjusted after measuring the signal at all endpoints.

For details, see the reference sources in "Recommended Reading" on page 19-19 for suggestions on transmission line termination.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the printed circuit board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the VDDEXT and VDDINT pins of the package as shown in Figure 19-5. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Connect the power plane to the power supply pins directly with minimum trace length. A ground plane should be located near the component side of the board to reduce the distance that ground current must travel through vias. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced.

V_{DDINT} is the highest frequency and requires special attention. Two things help power filtering above 100 MHz. First, capacitors should be physically small to reduce the inductance. Surface-mount capacitors of size 0402 give better results than larger sizes. Secondly, lower values of

capacitance raises the resonant frequency of the LC circuit. While a cluster of 0.1mF is acceptable below 50 MHz, a mix of 0.1, 0.01, 0.001mF and even 100 pF is preferred in the 500 MHz range.

Note that the instantaneous voltage on both internal and external power pins must at all times be within the recommended operating conditions as specified in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*. Local "bulk capacitance" (many microfarads) is also necessary. Although all capacitors should be kept close to the power consuming device, small capacitance values should be the closest. Larger values may be placed further from the chip.

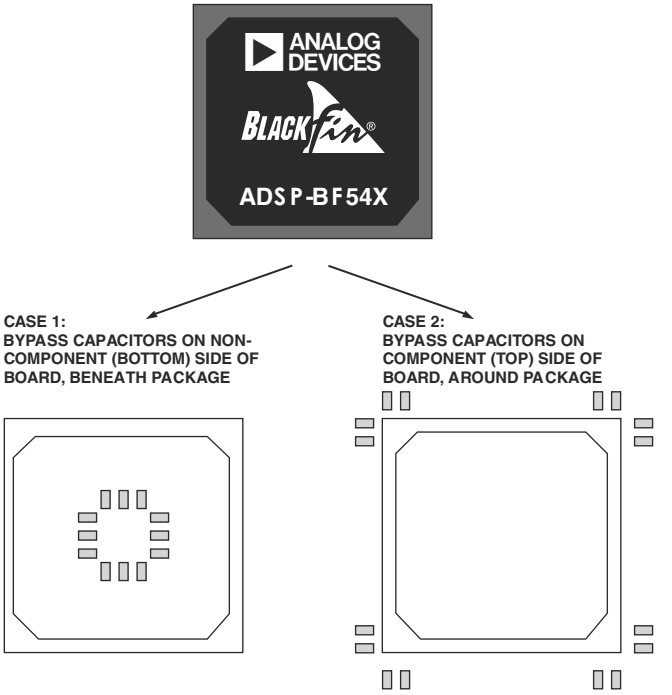


Figure 19-5. Bypass Capacitor Placement

5 Volt Tolerance

Outputs that connect to inputs on 5 V devices can float or be pulled up to 5 V. Only the few pins listed as 5 V tolerant in *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* should be subjected to 5 volts. Current limiting resistors are not sufficient to maintain long-term reliability. Level shifters are required on all other Blackfin pins to keep the pin voltage at or below absolute maximum ratings.

Resetting the Processor

The reset pin requires a monotonic rise and fall. Therefore the pin should not be connected directly to an R/C time delay because such a circuit could be noise-sensitive. In addition to the hardware reset mode provided through the `RESET` pin, the processor supports several software reset modes.

Recommendations for Unused Pins

Most often, there is no need to terminate unused pins, but the handful that do require termination are listed at the end of the pin list description section of *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

If the real-time clock is not used, `RTXI` should be pulled low. Also note that unused peripherals may have separate power connections. These should be driven to the specified value.

Peripheral specific power pins require power and ground even when the peripheral is not used.

Programmable Outputs and Pin Multiplexing

During power-up, each GPIO pin is set to an input and any pins used in the system as an output should be connected to a pullup or pulldown resistor to maintain the desired state.

This would be particularly important in motor drive applications. It is also important for UART TX and $\overline{\text{RTS}}$, CAN TX, SPI and serial TWI, ATAPI and other communications interfaces.

Boot Modes that use HWAIT require a pullup or pulldown resistor on PB11 on the ADSP-BF54x processor processors. HWAIT is driven both high and low during all boot cycles and may cause contention or unwanted values if also used as a GPIO.

After the boot cycle, GPIO pins may already be set to input or output depending on ADSP-BF54x processor family number and the boot cycle chosen. The I/O/GPIO muxing of all pins may need to be reprogrammed to support the users application. Care should be taken for compatibility of function and state, before boot, during boot, and during application pin usage.

Test Point Access

The debug process is aided by test points on signals such as CLKOUT or SCLK, bank selects, PPICLK, and $\overline{\text{RESET}}$. If selection pins such as boot mode are connected directly to power or ground, they are inaccessible under a BGA chip. Use pull-up and pull-down resistors instead.

Oscilloscope Probes

When making high speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state-of-the-art, high-frequency digital circuit design. It is an excellent source of information and practical ideas. Topics covered in the book include:

- High-speed properties of logic gates
- Measurement techniques
- Transmission lines
- Ground planes and layer stacking
- Terminations
- Vias
- Power systems
- Connectors
- Ribbon cables
- Clock distribution
- Clock oscillators

Consult your CAD software tools vendor. Some companies offer demonstration versions of signal integrity software. Simply by using their free software, you can learn:

- Transmission lines are real
- Unterminated printed circuit board traces ring and have overshoot and undershoot
- Simple termination controls signal integrity problems

20 NAND FLASH CONTROLLER

The ADSP-BF54x Blackfin processors provide a NAND flash controller (NFC) interface. The NFC on ADSP-54x processors provides the hardware support for the combination of hardware and software necessary to interface a processor with NAND flash devices. The NFC provides device access timing control and hardware error checking.

This chapter includes the following sections:

- "Overview" on page 20-2
- "Interface Overview" on page 20-4
- "Description of Operation" on page 20-5
- "Functional Description" on page 20-8
- "Programming Model" on page 20-16
- "NFC Registers" on page 20-18
- "NFC Programming Examples" on page 20-30

Overview

The NFC provides the following hardware features:

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

NAND flash devices provide high-density, low-cost memory. However, NAND flash devices also have long random access times, invalid blocks, and lower reliability over device lifetimes.

Because of these characteristics, NAND flash is often used for read-only code storage. In this case, all processor code can be stored in NAND flash and then transferred to a faster memory (such as SDRAM or SRAM) before execution.

Another common use of NAND flash is for storage of multimedia files or other large data segments. In this case, a software file system may be used to manage the reading and writing of the NAND flash device.

The file system selects memory segments for storage with the goal of avoiding bad blocks and equally distributing memory accesses across all address locations.

Bad block management includes both initial bad block detection and acquired bad block mapping. NAND flash devices contain bad blocks that are marked by the manufacturer. Software reads the bad block information, creates a table of bad block locations, and prevents use of the bad blocks. As additional blocks corrupt over time, they can be detected by the hardware and added to the bad block table by software. Software must provide bad block management, wear-leveling functions, and error correction. (See "NFC Error Detection" on page 20-12 for details on error correction.)

When NAND flash is used for read/write data storage, software wear-leveling is required. Wear-leveling increases the life span of NAND flash by generating an evenly distributed number of program and erase operations across the entire memory space. Software does this by translating logical addresses into different physical addresses for each write.

Interface Overview

Figure 20-1 shows the NFC interface.

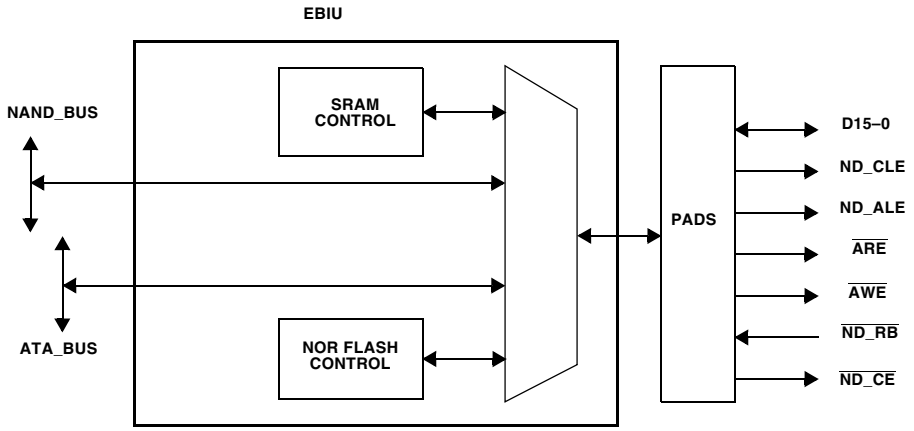


Figure 20-1. NFC Interface Block Diagram

The port pins used for NFC are shown in Table 20-1. The D15-0 bus, ND_CLE, ND_ALE, $\overline{\text{ARE}}$, and $\overline{\text{AWE}}$ pins are shared with the asynchronous memory controller. In addition, the data bus D15-0 is also shared with the ATAPI peripheral.

Table 20-1. NFC External Interface

Signal Name	Function	Default	Direction
D15–0	Data and Commands Bus	low	I/O
ND_CLE	Command Latch Enable	low	O
ND_ALE	Address Latch Enable	low	O
$\overline{\text{ARE}}$	Read Enable	high	O
$\overline{\text{AWE}}$	Write Enable	high	O
$\overline{\text{ND\_RB}}$	Ready/nBusy Request		I
$\overline{\text{ND\_CE}}$	Chip Enable	high	O

Description of Operation

The following sections describe the operation of the NAND flash controller.

Internal Bus Interfaces

The NFC interfaces to both the PAB and DAB buses on ADSP-BF54x processor Blackfin processors.

Page reads and page writes occur over DAB. The DAB interface consists of two separate 4-word FIFOs, one for page reads and one for page writes. Each FIFO is 32-bits wide in 32-bit DMA mode. Page reads and page writes cannot be triggered at the same time.

All other accesses occur over PAB. PAB accesses always go through the NFC write buffer. In 8-bit mode, this buffer is 8 bits wide, and, in 16-bit mode, this buffer is 16 bits wide. In both modes, it is 4 words deep. Software must prevent overflow of the buffer. Write buffer entries are not removed until the access is completed on the external interface. In the case of a read data request, the entry is not removed until the returned data is read from the `NFC_READ` register. After the fourth write to the write buffer, software must poll `WB_FULL` or `WB_EMPTY` in `NFC_STAT` to determine when there is additional space in the write buffer or use the `WB_EDGE` interrupt to detect when the write buffer has emptied.

After reset, the PAB write buffer has priority over the DAB FIFOs for access to the NFC external interface. If a page access is initiated while there are transfers in the write buffer, the page access does not start until the write buffer is empty. Likewise, once a page access starts, transfers in the write buffer do not begin until the page access is complete.

Bus Access Types

The NFC supports 8-bit or 16-bit NAND flash devices. PAB accesses cause only one transfer per bus access. For DAB access, the NFC automatically breaks up 32-bit DAB transfers into multiple NAND flash access cycles. Table 20-2 describes all the valid access types for both 8- and 16-bit devices as well as the number of NAND flash accesses it takes to complete the transaction.

Table 20-2. NFC Accesses

Bus	Bus Width	NAND Flash Width	NAND Flash Access Cycles Required
PAB	16-bit	8-bit	1
PAB	16-bit	16-bit	1
DAB	32-bit	8-bit	4
DAB	32-bit	16-bit	2

Access Timing

The NFC provides configurable access timing control for both read and write transactions through the NFC_CTL register.

The write enable pulse width (t_{WP}) is the WR_DLY + 1 SCLK. The WR_DLY selection should be configured such that:

$t_{WP} >= \text{Max}\ (t_{WPmin}\ ,\ (t_{CS} - 1\ \text{SCLK})\ )$

where t_{WP} is the time for which $\overline{\text{AWE}}$ is driven low, t_{WPmin} is the minimum write pulse duration from the NAND flash data sheet, and t_{CS} is the chip enable setup time from the NAND flash data sheet. See NAND Flash Controller Interface Timing in the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Likewise, the setup time for read data is configurable by changing RD_DLY in the NFC_CTL register. The RD_DLY selection should be configured such that:

$t_{RP} > Max\ (\ t_{RPmin}\ ,\ t_{REAmax}\ ,\ (\ t_{CEAmax} - 1\ SCLK)\ )$

where t_{RP} is the time for which $\overline{ARE}$ is driven low, t_{RPmin} is the minimum read pulse duration from the NAND flash data sheet, t_{REAmax} is the maximum read enable access time from the NAND flash data sheet, and t_{CEAmax} is the maximum chip enable access time from the NAND flash data sheet. See NAND Flash Controller Interface Timing in the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Pin Sharing

The NFC shares the ADSP-BF54x processor processors' pins with the AMC and ATAPI blocks. There is an asynchronous pin control module (APCM) that controls and arbitrates the asynchronous interface between the AMC, NAND, and ATA controllers. When an NFC transfer starts, the NFC requests the pins. Once the pins are granted, the NFC performs multiple transfers before releasing the pins. If the transfer is from the write buffer, NFC retains the pins until the write buffer is empty. If the transfer is a page access, the NFC performs eight external bus cycles, then checks to see if the AMC requires the pins. If the AMC does require the pins, NFC releases them. Otherwise, the NFC continues conducting transfers until the page is complete or an AMC request occurs.

Functional Description

The following sections describe the function of the NAND flash controller. NFC operation include:

- "Page Write" on page 20-8
- "Page Read" on page 20-10
- "Additional Operations" on page 20-11
- "Write Protection" on page 20-12
- "Chip Enable Don't Care" on page 20-12
- "NFC Error Detection" on page 20-12
- "NFC SmartMedia Support" on page 20-16

Page Write

To store data in NAND flash, first write the program command to the NFC_CMD register. Then, write a sequence of address bits to the NFC_ADDR register. For example, a 1Gbit x8 small page NAND flash device, consisting of 512 bytes per page, 32 pages per block and 8192 blocks requires 27 address bits in order to address the full range of memory. In this case, address bits [7:0] are written to address the column within the page to access. This is then followed by writing address bits[16:9], [24:17] and finally [26:24]. Note that for small page NAND flash devices, address bit [8] is generated automatically by the NAND flash device. Once the DMA channel has been configured, the next step is to set the page write start bit in the NFC_PGCTL register. This initiates DMA transfers to complete the page write. After writing all of the data, software can append the ECC values from the ECC registers to store them in the spare area of the NAND flash. Finally, the page program confirm command is written to NFC_CMD to initiate the NAND flash programming process. The NAND flash

asserts $\overline{\text{ND\_RB}}$ until the page is completely programmed. At that time, the write status bit in the NAND flash device may be checked. Figure 20-2 shows the timing of a NAND flash write access for a device requiring only three address cycles.

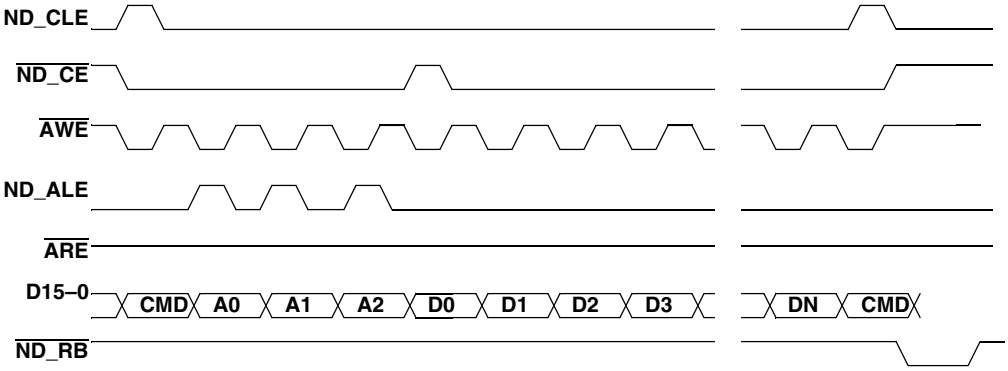


Figure 20-2. NAND Flash Program Operation

Page Read

To read data from NAND flash, first write the read command to the NFC_CMD register. Then write a sequence of address bits to the NFC_ADDR register. For a 1Gbit x8 small page NAND flash device, consisting of 512 bytes per page, 32 pages per block and 8192 blocks, 27 address bits are required in order to address the full range of memory. Address bits [7:0] are written first in order to address the column to access. This is then followed by the address bits [16:9], [24:17] and [26:24]. Not that for small page devices [A8] is generated automatically by the NAND flash device and is determined by the read command that is issued prior to the address cycles. Once all the address cycles have been issued the NAND flash device becomes busy and software should wait for the rising edge of $\overline{\text{ND\_RB}}$, indicating that the requested data is available. Once the DMA channel has been configured, set the page read start bit in NFC_PGCTL. This initiates the DMA transfers for a page read. As each read occurs, new ECC values are calculated for each 256 or 512 byte page.

When the page read is complete, the core may complete final data read requests to obtain the stored ECC values which were written in the spare area when the page was programmed. Software can compare this to the new ECC values to determine if any bit errors have occurred. Figure 20-3 shows the timing of a NAND flash read access for a device requiring only three address cycles.

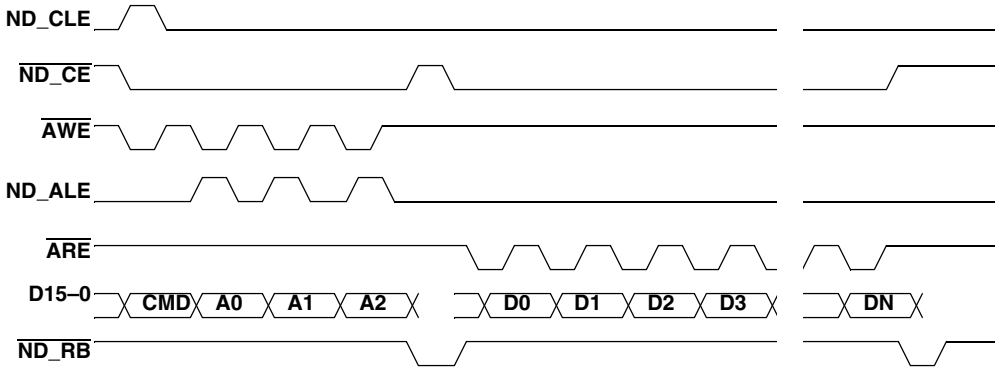


Figure 20-3. NAND Flash Read Operation

Additional Operations

The core may execute data read and write transactions directly without the requirement of DMA. Commands must be written to NFC_CMD, addresses must be written to NFC_ADDR. For data write transactions, the data to be written to the NAND flash must go via the NFC_DATA_WR register. Data reads are requested by first writing to NFC_DATA_RD in order to issue the read transaction on the NFC interface and then reading back the received data from NFC_READ after the RD_RDY interrupt has been generated.

To check that an operation is complete, $\overline{\text{ND\_RB}}$ may be polled in the NFC_STAT register or used to trigger an interrupt. Software must always poll or wait for the $\overline{\text{ND\_RB}}$ before performing an operation.

For 8-bit NAND flash devices, only the lower 8 bits of the NFC_CMD, NFC_ADDR, NFC_DATA_WR, and NFC_READ registers are valid; the higher bytes are ignored.

All SLC NAND flash device operations are supported via writing to or reading from the various NFC registers. For example, erasing a block on the NAND flash requires the issuing of a specific block erase command followed by a number of address cycles followed by a block erase confirmation command. More advanced operations such as cache program operations are also supported.

See NAND flash device data sheets for examples of these operations.

Write Protection

NAND flash devices require a write protection input signal (nWP) to prevent inadvertent write or erase operations. A GPIO can be used for this purpose.

Chip Enable Don't Care

Some NAND flash devices ignore the read enable, write enable, command latch enable, and address latch enable control signals when chip select is deasserted during page reads and page programs. These devices are called chip enable don't care (CEDC) NAND flash devices. This is the only type of device supported by the NFC.

NFC Error Detection

The NFC error checking and correction (ECC) logic can detect one bit of correctable error or multiple bits of non-correctable error. The NFC employs a Hamming code algorithm, which generates two sets of parity bits for every 256 bytes of data. For 512-byte pages, the page is split into two halves, and separate ECC values are calculated for each half.

For every 256 bytes of data, 22 bits of ECC parity data are generated as follows:

```
P1 = D[1] ^ D[3] ^ D[5] ^ D[7] ^ D[9] …^ D[2047];

P2 = D[2] ^ D[3] ^ D[6] ^ D[7] ^ D[10] ^ D[11] … ^ D[2042] ^
D[2043] ^ D[2046] ^ D[2047];

P4 = D[4] ^ D[5] ^ D[6] ^ D[7] ^ D[12] ^ D[13] ^ D[14] ^ D[15] ^
D[20] ^ D[21] ^ D[22] ^ D[23] … ^ D[2044] ^ D[2045] ^ D[2046] ^
D[2047];

P8 = D[8] ^ D[9] ^ D[10] ^ D[11] ^ D[12] ^ D[13] ^ D[14] ^ D[15]
^ D[24] ^ D[25] ^ D[26] D[27] ^ D[28] ^ D[29] ^ D[30] ^ D[31] … ^
D[2040] ^ D[2041] ^ D[2042] ^ D[2043] ^ D[2044] ^ D[2045] ^
D[2046] ^ D[2047];

…
…

P1’ = D[0] ^ D[2] ^ D[4] ^ D[6] ^ D[8] … ^ D[2046];

P2’ = D[0] ^ D[1] ^ D[4] ^ D[5] ^ D[8] ^ D[9] … ^ D[2040] ^
D[2041] ^ D[2044] ^ D[2045];

P4’ = D[0] ^ D[1] ^ D[2] ^ D[3] ^ D[8] ^ D[9] ^ D[10] ^ D[11] ^
D[16] ^ D[17] ^ D[18] ^ D[19] … ^ D[2040] ^ D[2041] ^ D[2042] ^
D[2043];

…
…
```

In this way, P1, P2, P4, P8, P16, P32, P64, P128, P256, P512, and P1024 as well as P1', P2', P4', P8', P16', P32', P64', P128', P256', P512', and P1024' are calculated, producing a total of 22 parity bits for each 256 bytes (2048 bits) of data.

The NFC writes this 22-bit ECC value into the NFC_ECCx registers. Software can store these values in the spare area of the NAND flash device for later comparison. When reading back data, the NFC automatically calculates new ECC values from the received data. Software can generate error syndromes by exclusive OR'ing the stored and newly calculated ECC values.

Error Analysis

Analyzing the ECC values lets you determine the error syndrome. The resulting error syndromes indicate what type of data errors have occurred.

For example, when a 256 byte page is read back, ECC0(stored) contains the parity bits stored read from the spare area. ECC1(stored) contains the parity' bits read from the spare area. Similarly, ECC0(calculated) and ECC1(calculated) contain the newly calculated parity and parity' bits, respectively. To interpret the ECC values, software generates the following error syndromes:

```
syndrome0[21:0] = {ECC0calculated[10:0],ECC1calculated[10:0]} ^
{ECC0stored[10:0],ECC1stored[10:0]}

syndrome1[10:0] = ECC0calculated[10:0] ^ ECC0stored[10:0]

syndrome2[10:0] = ECC0calculated[10:0] ^ ECC1calculated[10:0]

syndrome3[10:0] = ECC0stored[10:0] ^ ECC1stored[10:0]

syndrome4[10:0] = syndrome2[10:0] ^ syndrome3[10:0]
```

Syndrome 0 indicates whether there is an error in the data. Syndrome 4 indicates whether the error is a 1-bit correctable error. Syndrome 1 indicates the bit location of any 1-bit errors. After calculating these syndromes, software must examine their values and take the appropriate actions.

- If Syndrome 0 is 0x000, the data is valid and no actions are required.
- If Syndrome 0 has exactly 11 bits that are 1 and Syndrome 4 is 0x7FF, there is a 1-bit correctable error. Syndrome 1 gives the failing bit number. For example, if Syndrome 1 is 46, bit 6 in the sixth word transferred needs to be inverted.
- If Syndrome 0 has only 1 bit that is 1, there is an error in the ECC data itself. No action is required, since ECC data is discarded after each page read, but no error checking can be done.
- If Syndrome 0 has any other value, there is a multiple-bit, unrecoverable error. Software should mark the block containing this page as a bad block.

Examples of possible Syndrome 0 values are shown in Table 20-3.

Table 20-3. ECC Syndrome Examples

Syndrome 0	Type of Value	Meaning	Action Required
0x00 0000	All zero	No error in data	None
0x2C CA66	Exactly 11 bits are 1, each parity and parity' pair is 1 & 0 or 0 & 1	1-bit correctable error	Correct error
0x00 0040	Only 1 bit is 1	ECC data was incorrect	None
0x06 B35A	Random data	More than 1-bit error, non-correctable error	Discard data, mark bad block

Large Page Size Support

Page sizes larger than 512 bytes can be supported by NFC as long as they require only 1-bit error correction per 512 Bytes. For example, a 2K byte page can be accessed by treating it as four 512-byte pages. The page program and page reads must be conducted as four 512-byte accesses, and the ECC values for each 512 bytes of data must be read back from the ECC registers and then temporarily stored. The ECC registers must be reset before the next 512 bytes are transferred. Once ECC values from all four 512-byte pages are calculated, they are typically written into the NAND flash spare area for a page write or compared to those in the spare area for a page program.

NFC SmartMedia Support

NAND flash and SmartMedia devices have nearly identical interfaces. The main difference is that SmartMedia devices are removable, and, therefore, require card insertion, card ejection and write protection signals. On ADSP-BF54x processor Blackfin processors, these features can be supported using GPIOs.

Programming Model

The following sections describe the NAND flash controller's programming model.

Before using the NFC, pins with GPIO functions must be configured to select the NFC functionality. This causes a rising edge detect on $\overline{\text{ND\_RB}}$, which must be cleared before beginning NFC programming sequences.

To conduct a page read, the core may use the following procedure:

1. The core writes to the appropriate DMA registers to enable the NFC DMA channel for receive mode and to configure the correct number of transfers for a single page.
2. The core sets up the appropriate configuration by writing the NFC_CTL register.
3. The core clears the NFC_ECCx registers by setting the ECC_RST bit in the NFC_RST register.
4. The core writes the page read commands to NFC_CMD register and the page addresses to the NFC_ADDR register (maximum of four writes at a time).
5. The core waits for a rising edge detection on $\overline{\text{ND\_RB}}$.
6. The core sets the page read start bit in the NFC_PGCTL register.
7. When the DMA generates an interrupt on completion, the core reads the remaining spare bytes.
8. The core compares ECC information stored in the spare bytes to the ECC register values calculated during the page read.
9. If there is an ECC error, the core must correct the corrupted data.

To conduct a page write, the core may use the following procedure:

1. The core writes to the appropriate DMA registers to enable the NFC DMA channel for transmit mode and to configure the correct number of transfers for a single page.
2. The core sets up the appropriate configuration by writing the NFC_CTL register.
3. The core clears the NFC_ECCx registers by setting the ECC_RST bit in the NFC_RST register.

4. The core writes the page write commands to NFC_CMD register and the page addresses to the NFC_ADDR register (maximum of 4 writes at a time).
5. The core waits for the write buffer to be empty by either polling the status bit or waiting for the WB_EDGE interrupt.
6. The core sets the page write start bit in the NFC_PGCTL register.
7. When the DMA generates an interrupt on completion, the WR_DONE bit should be checked to verify the last transfer is complete, then the core reads the ECC register values and writes those values to the spare bytes of the page.
8. The core writes the page program confirm command to the NFC_CMD register.
9. The core waits for the write buffer to empty and for a subsequent rising edge detection on $\overline{\text{ND\_RB}}$.

NFC Registers

The NFC has a group of memory-mapped registers (MMRs) that regulate its operation. These registers are listed in Table 20-4 on page 20-19.

Descriptions and bit diagrams for each of these MMRs are provided in the following sections. The NFC MMRs start at a base address of 0xFFC0 3B00.

The NFC contains control, status, interrupt and ECC registers at address offsets 0x00]0x2C. The NFC also contains write-only registers at address offsets 0x40]0x4C that insert commands, address, or data access requests into a write buffer.

The NFC_ECCx and NFC_COUNT registers should not be read while an access to NAND flash is happening on the EBIU. Otherwise, the registers may be updating during a read and coherency of the register bits is not guaranteed.

Table 20-4 lists all of the NFC memory-mapped registers.

Table 20-4. NFC Memory-Mapped Registers

Address	Register Name	Description
0xFFC0 3B00	NFC_CTL	"NFC Control Register (NFC_CTL)" on page 20-20
0xFFC0 3B04	NFC_STAT	"NFC Status Register (NFC_STAT)" on page 20-21
0xFFC0 3B08	NFC_IRQSTAT	"NFC Interrupt Status Register (NFC_IRQSTAT)" on page 20-22
0xFFC0 3B0C	NFC_IRQMASK	"NFC Interrupt Mask Register (NFC_IRQMASK)" on page 20-23
0xFFC0 3B10	NFC_ECC0	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B14	NFC_ECC1	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B18	NFC_ECC2	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B1C	NFC_ECC3	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B20	NFC_COUNT	"NFC Count Register (NFC_COUNT)" on page 20-25
0xFFC0 3B24	NFC_RST	"NFC Reset Register (NFC_RST)" on page 20-25
0xFFC0 3B28	NFC_PGCTL	"NFC Page Control Register (NFC_PGCTL)" on page 20-26
0xFCC0 3B2C	NFC_READ	"NFC Read Data Register (NFC_READ)" on page 20-26
0xFFC0 3B40	NFC_ADDR	"NFC Address Register (NFC_ADDR)" on page 20-27
0xFFC0 3B44	NFC_CMD	"NFC Command Register (NFC_CMD)" on page 20-28
0xFFC0 3B48	NFC_DATA_WR	"NFC Data Write Register (NFC_DATA_WR)" on page 20-29
0xFFC0 3B4C	NFC_DATA_RD	"NFC Data Read Register (NFC_DATA_RD)" on page 20-29

NFC Control Register (NFC_CTL)

The NFC_CTL register (see Figure 20-4) contains timing and mode configuration fields. The read strobe delay (RD_DLY) and write strobe delay (WR_DLY) fields extend the $\overline{\text{ARE}}$ and $\overline{\text{AWE}}$ strobes, respectively, by the specified number of cycles. If no extension is specified, $\overline{\text{ARE}}$ and $\overline{\text{AWE}}$ assert for a single SCLK cycle. The NAND data width (NWIDTH) bit selects the data bus width size of the external NAND flash device. The page size (PG_SIZE) bit determines where the ECC data values are written. For a 256-byte page, ECC values are always calculated in NFC_ECC0 and NFC_ECC1. For a 512-byte page, the first ECC value is calculated in NFC_ECC0 and NFC_ECC1 while the next ECC value is calculated in NFC_ECC2 and NFC_ECC3.

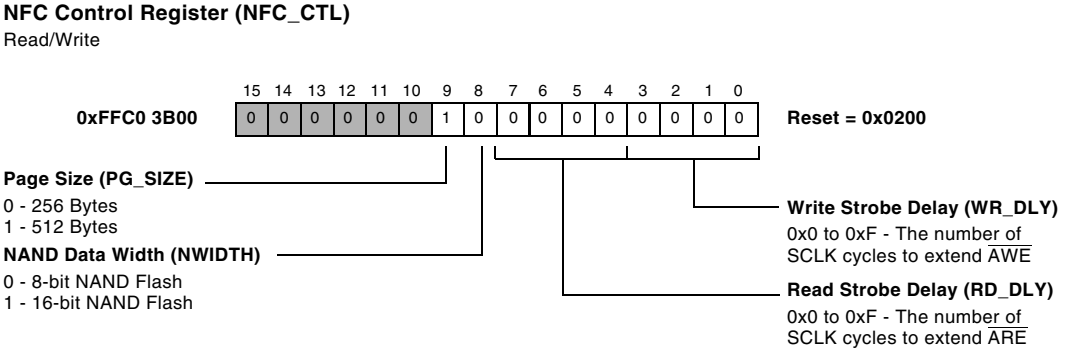


Figure 20-4. NFC Control Register

NFC Status Register (NFC_STAT)

The NFC_STAT register (see Figure 20-5) contains status information. The NBUSY bit contains the synchronized value of the $\overline{\text{ND\_RB}}$ pin. The write buffer empty (WB_EMPTY) and write buffer full (WB_FULL) status bits contain write buffer status information. When WB_FULL is set, writes to any write buffer register are ignored and cause the WB_OVF bit in the NFC_IRQSTAT register to be set. The page write pending (PG_WR_STAT) and page read pending (PG_RD_STAT) bits show indicate that a page write (or read) is started and not completed.

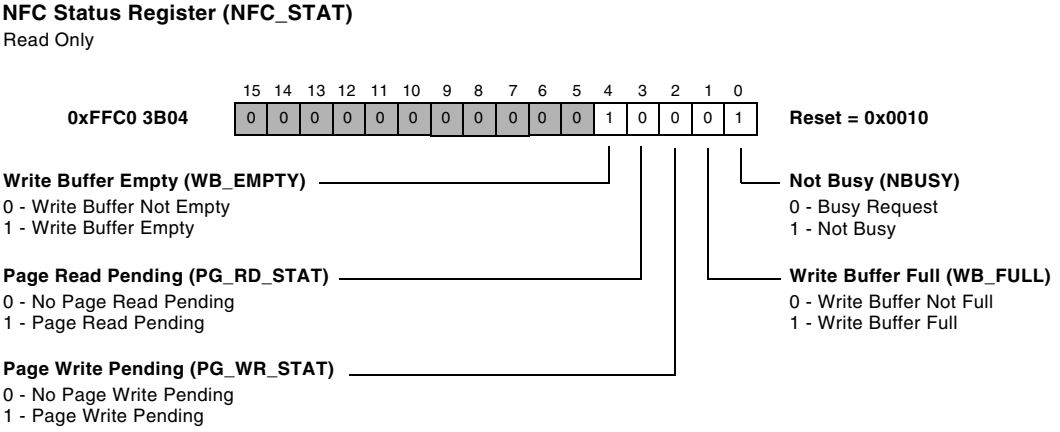


Figure 20-5. NFC Status Register

As soon as the $\overline{\text{ND\_RB}}$ signal has been enabled via PORTJ_FER and the signal is sampled as high. The NBUSY bit is set in NFC_STAT and the NBUSYIRQ interrupt is generated

NFC Interrupt Status Register (NFC_IRQSTAT)

The NFC_IRQSTAT register (see Figure 20-6) reports the status of additional NFC interrupt sources. All bits in this register are write-1-to-clear (W1C). The NBUSYIRQ sticky bit is asserted when a rising edge is detected on the $\overline{\text{ND\_RB}}$ signal. This bit must be cleared (W1C) before starting a new access. The WB_OVF bit is asserted when the write buffer overflows and indicates an error condition. The write buffer edge detect (WB_EDGE) bit is set when the write buffer transitions from not empty to empty. The read data ready (RD_RDY) bit indicates that a read data command has completed and that data is available for reading from the NFC_READ register. The page write done (WR_DONE) bit indicates a completed page write and that the last access in the page was transferred on the external bus.

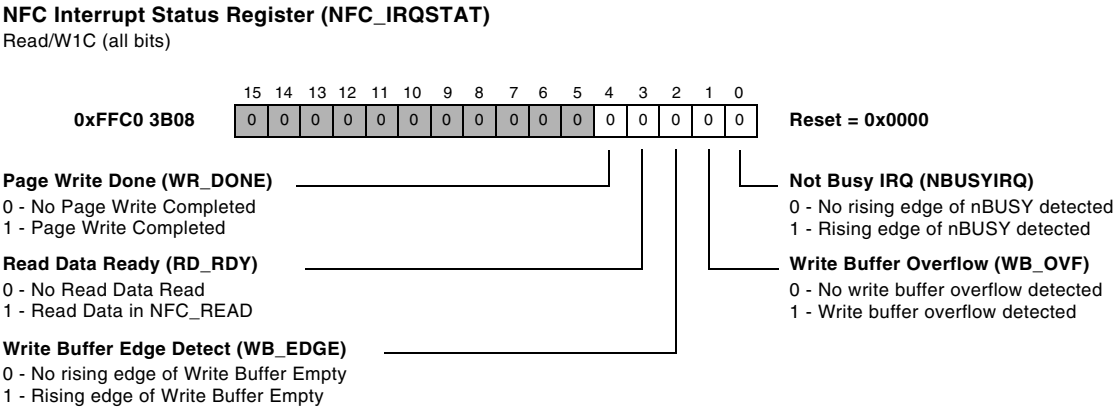


Figure 20-6. NFC Interrupt Status Register

As soon as the $\overline{\text{ND\_RB}}$ signal has been enabled via PORTJ_FER and the signal is sampled as high. The NBUSY bit is set in NFC_STAT and the NBUSYIRQ interrupt is generated

NFC Interrupt Mask Register (NFC_IRQMASK)

The NFC_IRQMASK register (see Figure 20-7) contains individual mask bits for each NFC interrupt source. After masking, the bits are OR'ed together and routed to the system interrupt controller.

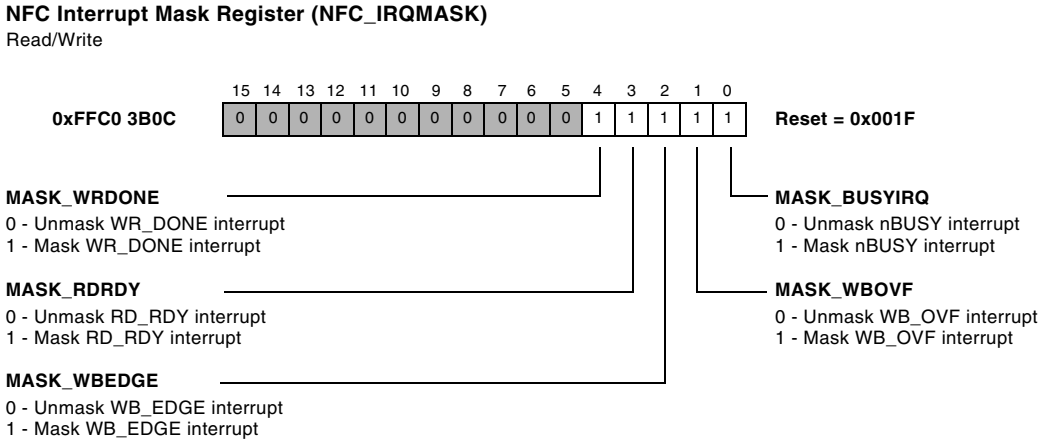


Figure 20-7. NFC Interrupt Mask Register

NFC ECC Registers (NFC_ECCx)

The NFC_ECCx registers (see Figure 20-8) contain the 22-bit ECC parity values calculated for data read from or written to the NAND flash device. When data is written, the processor must store these values in the spare area of the NAND flash device. When data is read, the ECC values are calculated for comparison with the values stored in the spare area.

The four 16-bit ECC registers are used to hold the ECC data as it is calculated by the ECC logic. The registers NFC_ECC0 and NFC_ECC1 are used to hold the 22-bit ECC value for the first 256 bytes of the page. For 512-byte pages, the registers NFC_ECC2 and NFC_ECC3 hold the 22-bit ECC value for the second half-page (256 bytes). The page size is configured in NFC_CTL register.

The values in the ECC registers are updated on every cycle that data is transferred. They are not updated when spare area bytes are read or written. The registers NFC_ECC0 and NFC_ECC1 are valid after the transfer of the 256th byte in a page. the registers NFC_ECC2 and NFC_ECC3 are valid after the transfer of the 512th byte in a page.

Note that the ECC registers are 16 bits each. When writing the ECC value to an 8-bit NAND flash device, the lower 8 bits must be written first, followed by the upper 8 bits.

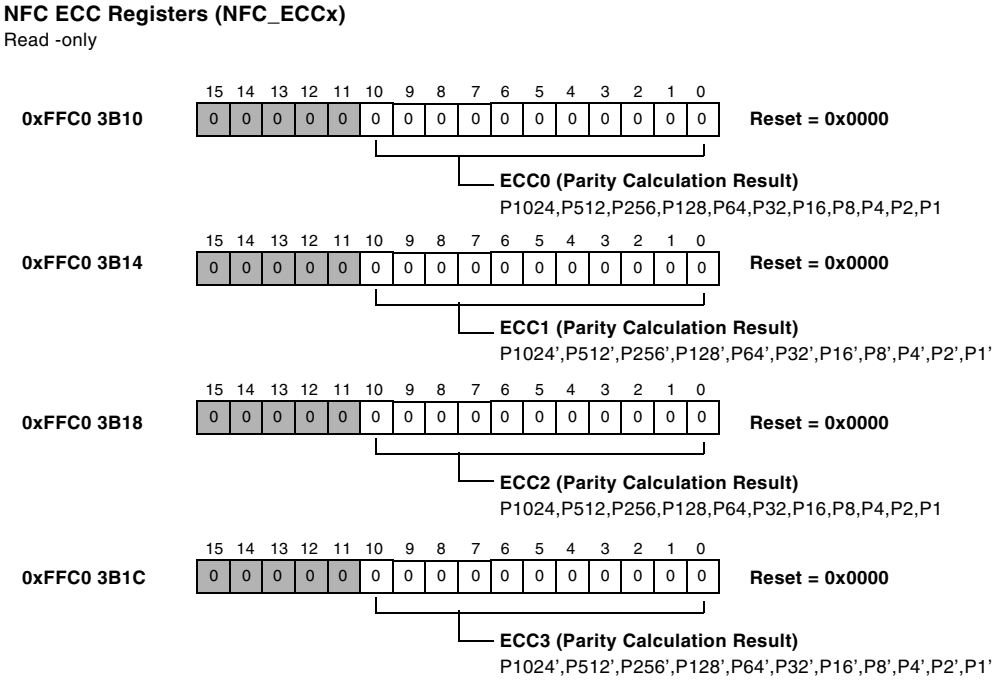


Figure 20-8. NFC ECC Registers

NFC Count Register (NFC_COUNT)

The NFC_COUNT register (see Figure 20-9) reports the number of bytes transferred in the current page. The count starts at 1 and increments up to 512 for a 512-byte page. This register is used primarily for debugging purposes. The counter is reset when the ECC_RST bit in the NFC_RST register is set.

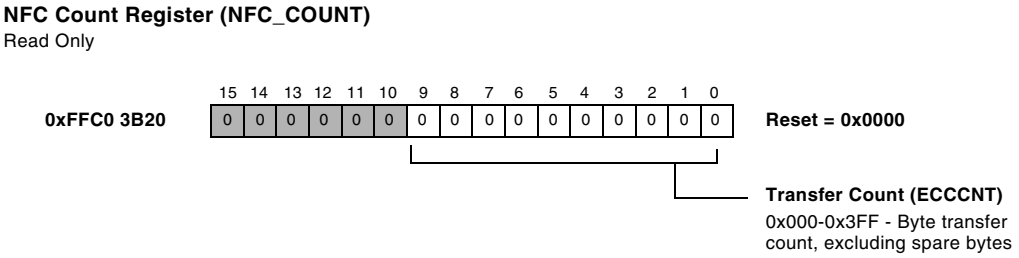


Figure 20-9. NFC Count Register

NFC Reset Register (NFC_RST)

The NFC_RST register (see Figure 20-10) allows software to reset the ECC registers and the NFC counters. This register must be written before each page is transferred to generate the correct ECC register values. The ECC reset bit is automatically cleared by the NFC on completion of the reset.

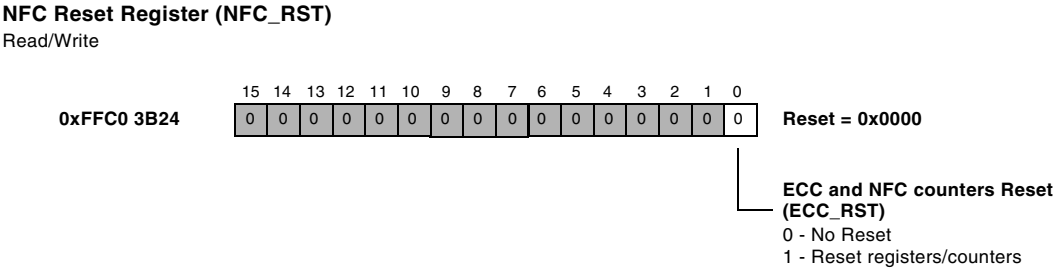


Figure 20-10. NFC Reset Register

NFC Page Control Register (NFC_PGCTL)

The NFC_PGCTL register (see Figure 20-11) allows the processor to initiate page reads or writes. All bits in the register are write only. The page data is always transferred using the DAB bus. When either a page read or page write is pending, page read start (PG_RD_START) and page write start (PG_WR_START) are ignored.

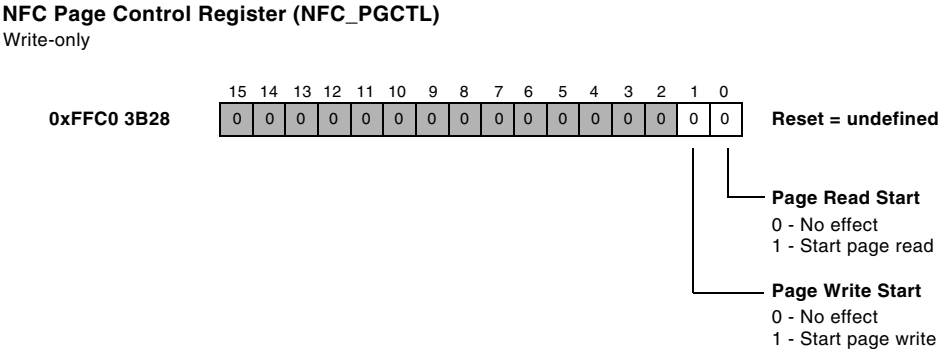


Figure 20-11. NFC Page Control Register

NFC Read Data Register (NFC_READ)

The NFC_READ register (see Figure 20-12) contains read data returned from the NAND flash after a read is requested using the NFC_DATA_RD register. If NWIDTH is configured for 8 bits, only the eight LSB have valid data, otherwise all 16 bits have valid data. The RD_RDY status bit and interrupt indicate when new data is available for reading.

To prevent overflow of NFC_READ, the read data request is not removed from the write buffer until the returned data is read back from NFC_READ. As a result, no other commands, address or data are sent to the NAND flash while the read data request is active in the write buffer.

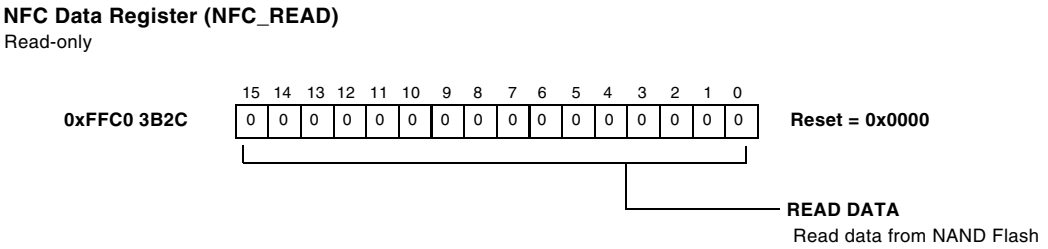


Figure 20-12. NFC Read Data Register

NFC Address Register (NFC_ADDR)

The NFC_ADDR register (see Figure 20-13) contains address bits to send to the NAND flash device. The number of address bits (8 or 16) sent to the NAND flash device is determined by the NWIDTH bit in the NFC_CTL register. Values written to this register are stored in the NFC write buffer.

If the user is connecting to a 16-bit NAND flash that requires 8-bit addresses, the user can program bits 0-7 of this register with the address and zero-out bits 8-15.

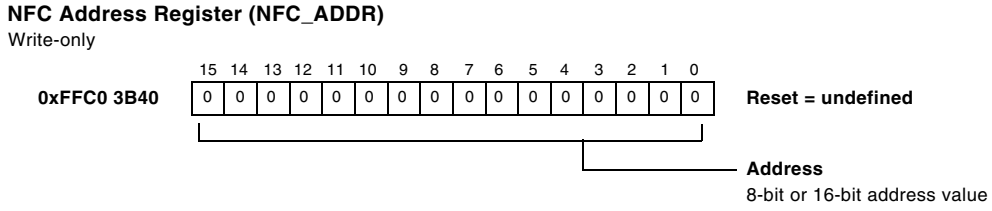


Figure 20-13. NFC Address Register

NFC Command Register (NFC_CMD)

The NFC_CMD register (see Figure 20-14) contains commands to write to the NAND flash device. The number of command bits (8 or 16) sent to the NAND flash device is determined by the NWIDTH bit in the NFC_CTL register. Values written to this register are stored in the NFC write buffer

If the user is connecting to a 16-bit NAND flash that requires 8-bit commands, the user can program bits 0-7 of this register with the command and zero-out bits 8-15.

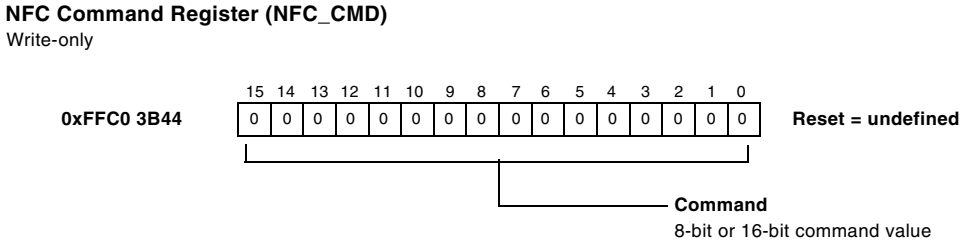


Figure 20-14. NFC Command Register

NFC Data Write Register (NFC_DATA_WR)

The NFC_DATA_WR register (see Figure 20-15) contains data to write to the NAND flash device. The number of data bits (8 or 16) sent to the NAND flash device is determined by the NWIDTH bit in the NFC_CTL register. Values written to this register are stored in the NFC write buffer.

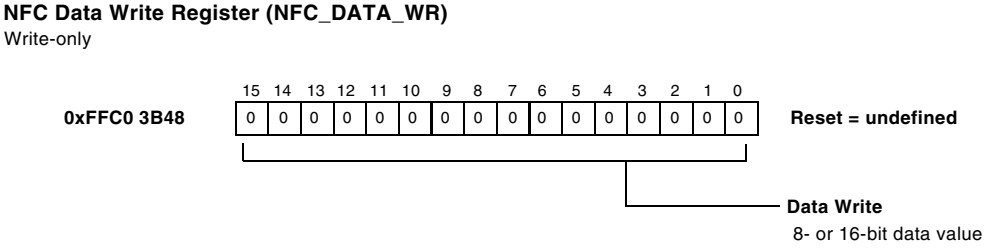


Figure 20-15. NFC Data Write Register

NFC Data Read Register (NFC_DATA_RD)

The NFC_DATA_RD register (see Figure 20-16) triggers a read request to the NAND flash device. The data written is ignored. The number of data bits (8 or 16) sent to the NAND flash device is determined by the NWIDTH bit in the NFC_CTL register. The read request from this register is stored in the NFC write buffer.

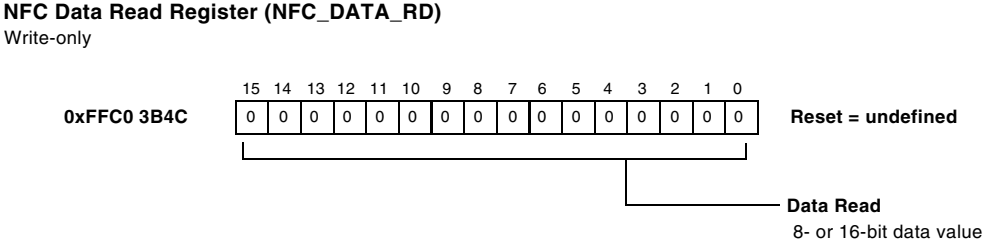


Figure 20-16. NFC Data Read Register

NFC Programming Examples

Listing 20-1 illustrates an example initialization sequence to enable the use of the NFC.

Listing 20-1. NFC Port Register Configuration

```
/* Bit macros for NFC Read and Write Strobe Delays */
#define SET_NFC_WR_STROBE(x)   ((x)&0xF)
#define SET_NFC_RD_STROBE(x)   (((x)&0xF)<<4)

/*********************************************************
 Mask out all NFC IRQs
*********************************************************/
P5.L  = lo(NFC_IRQMASK);
P5.H  = hi(NFC_IRQMASK);
R7.L  = MASK_WRDONE | MASK_RDRDY | MASK_WBEDGE | MASK_WBOVF |
MASK_BUSYIRQ;
w[P5] = R7.L;

/*********************************************************
 Configure port J NFC features
*********************************************************/
P5.L  = lo(PORTJ_FER);
P5.H  = hi(PORTJ_FER);
R7.L  = nPJ15 | nPJ14 | nPJ13 | nPJ12 | nPJ11 | nPJ10 | nPJ9 |
nPJ8 | nPJ7 |
        nPJ6 | nPJ5 | nPJ4 | nPJ3 | PJ2 | PJ1 | nPJ0;
w[P5] = R7.L;

/*********************************************************
 Configure port J MUX for NFC use
*********************************************************/
P5.L  = lo(PORTJ_MUX);
```

```
P5.H  = hi(PORTJ_MUX);
R7.L  = lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));
R7.H  = hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));
[P5] = R7;

/*********************************************************
 Configure NFC Control register
*********************************************************/
P5.L  = lo(NFC_CTL);
P5.H  = hi(NFC_CTL);
R7.L  = nPG_SIZE | nNWIDTH | SET_NFC_RD_STROBE(3) |
SET_NFC_WR_STROBE(3);
w[P5] = R7.L;

/*********************************************************
 Clear any IRQs that may be pending for the NFC.
*********************************************************/
P5.L  = lo(NFC_IRQSTAT);
P5.H  = hi(NFC_IRQSTAT);
R7.L  = WR_DONE | RD_RDY | WB_EDGE | WB_OVF | NBUSYIRQ;
w[P5] = R7.L;
ssync;

/*********************************************************
 Enable required NFC IRQs
*********************************************************/
P5.L  = lo(NFC_IRQMASK);
P5.H  = hi(NFC_IRQMASK);
R7.L  = nWR_DONE | nRD_RDY | nWB_EDGE | nWB_OVF | nNBUSYIRQ;
w[P5] = R7.L;
ssync;
```

Listing 20-2 illustrates one method on how to perform a page read from the NAND flash through core read transactions. This example assumes that only NFC wakeup interrupts are being used to bring the processor out of idle.

The attached NAND flash is a large page device that requires the issuing of 5 address cycles for a read operation, the page size of the device is 2048 bytes excluding the spare area. It is assumed that the NAND flash may not be in read mode already. For a byte read transaction to take place the NAND flash must first of all be configured for read mode. The address cycles must then be issued for the data that we wish to access. As it is a large page device the last address cycle is typically followed by a page read confirmation command. Once the NAND flash accepts the page read confirmation command the device enters a busy state while it transfers the data to be accessed from the main array into the read buffer where it can then be accessed through core read transactions.

The data is read from within 2 loops. The outer loop is executed 8 times in this example.

Outer loop count = NAND Page Size / NFC Page Size

The inner loop is configured for 256 bytes.

Inner loop count = NFC Page Size

Each execution of the inner loop reads one byte from the NAND flash by issuing a read transaction through the NFC_DATA_RD register. The received data is then read from the NFC_READ register and stored to a buffer in internal memory named “_Buffer”. At the end of every 256 byte block the 22-bit parity data is read from the NFC_ECC1 and NFC_ECC0 registers and stored to a second buffer in internal memory named “_CalculatedECC” before resetting the NFC ready for the next 256 byte block.

Upon completion of reading all 2048 bytes, the spare area is then read and stored at the bottom of the available 2112 byte buffer. This data would be used along with the newly calculated error correction parity data within the error correction routine to ensure all read data is correct.

Listing 20-2. Page Read through core read transactions

```
/*********************************************************
 Ensure the NFC write buffer is empty
*********************************************************/
P5.L  = lo(NFC_CTL);
P5.H  = hi(NFC_CTL);
_check_write_buffer_empty:
    R7  = w[P5 + lo(NFC_STAT - NFC_CTL)](z);
    CC = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty;
_check_write_buffer_empty.END:

/*********************************************************
 Issue Read Command (0x00) to NAND Flash
*********************************************************/
R7  = 0(x);
w[P5+ lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 In order to avoid a write buffer overflow error issue 3
 of the 5 address cycles to the NAND flash.
 *********************************************************/
R7 = 0(x);
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;

/*********************************************************
```

```
 Wait for write buffer to become empty
*********************************************************/
_check_write_buffer_empty_again:
    R6  = w[P5 + lo(NFC_STAT-NFC_CTL)](z);
    CC = bittst(R6, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty_again;
_check_write_buffer_empty_again.END:

/*********************************************************
 Issue the remaining 3 address cycles followed by the
 Read Confirmation command
*********************************************************/
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
R7  = 0x30(z);
w[P5+lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 Wait for the NFC not busy wakeup interrupt
*********************************************************/
_wait_for_ready:
    IDLE;
    R7 = w[P5+lo(NFC_IRQSTAT - NFC_CTL)](z);
    CC = bittst(R7, bitpos(NBUSYIRQ));
_wait_for_ready.END: IF !CC JUMP _wait_for_ready;
R7 = NBUSYIRQ(z);
w[P5+lo(NFC_IRQSTAT - NFC_CTL)] = R7;

/*********************************************************
 The page we wish to access is now ready to be read from
 the NAND flash. Set up the pointers to the data buffer
 and the buffer where we will store the calculated error
 correction parity data.
 *********************************************************/
```

```
P4.L = lo(_Buffer);
P4.H = hi(_Buffer);
P3.L = lo(_CalculatedECC);
P3.H = hi(_CalculatedECC);
P2 = (2048/256)(z);

LSETUP(_read_data_begin, _read_data_end) LC0 = P2;
P2= 256(z);

_read_data_begin: /* Outer loop */

    /* Reset the NFC */
    R7 = ECC_RST(z);
    w[P5 + lo(NFC_RST - NFC_CTL)] = R7;
    ssync;
    _wait_for_nfc_reset_completion:
        R7 = w[P5 + lo(NFC_RST - NFC_CTL)](z);
        CC = bittst(R7, bitpos(ECC_RST));
    _wait_for_nfc_reset_completion.END:
        if CC jump _wait_for_nfc_reset_completion;

    LSETUP(_read_nfc_page_begin, _read_nfc_page_end) LC1 = P2;
    _read_nfc_page_begin: /* Inner loop */
        w[P5+ lo(NFC_DATA_RD - NFC_CTL)] = R7;

        _wait_for_data_ready:
            IDLE;
            R7 = w[P5+ lo(NFC_IRQSTAT - NFC_CTL)];
            CC = bittst(R7, bitpos(RD_RDY));
            IF !CC JUMP _wait_for_data_ready;
        _wait_for_data_ready.END:
```

```
/*********************************************************
        The byte is now available in the NFC_READ register.
        We need to read the byte which results in the read
        transaction then being removed from the write buffer.
        We need to ensure that this transaction completes before
        clearing the IRQ

*********************************************************/
        R7 = RD_RDY(z);
        R6 = w[P5+ lo(NFC_READ - NFC_CTL)](z);
        ssync;
        w[P5+ lo(NFC_IRQSTAT - NFC_CTL)] = R7;
    _read_nfc_page_end: b[P4++] = R6;

    /* Read and store the error correction parity data */
    R7 = w[P5+ lo(NFC_ECC0 - NFC_CTL)](z);
    R6 = w[P5+ lo(NFC_ECC1 - NFC_CTL)](z);
    R6 <<= 11;
    R7 = R7 | R6;
_read_data_end: [P3++] = R7;
```

Listing 20-3 illustrates one method on how to perform a page program operation to the NAND flash through core write transactions. This example assumes that only NFC wakeup interrupts are being used to bring the processor out of idle.

The attached NAND flash is a large page device that requires the issuing of 5 address cycles for a program operation, the page size of the device is 2048 bytes excluding the spare area. It is assumed that the NAND flash may not be in program mode already.

For a byte write transaction to take place the NAND flash must first of all be configured for page program mode. The address cycles must then be issued followed by the page data. This is followed by a page program confirmation command. Once the NAND flash accepts the page program confirmation command the device enters a busy state while it transfers the data to be written into the main NAND flash array.

The data is written from within 2 loops. The outer loop is executed 8 times in this example.

Outer loop count = NAND Page Size / NFC Page Size

The inner loop is configured for the writing of 4 bytes per iteration and is executed 64 times in order to write a 256 byte block.

Inner loop count = NFC Page Size/4

At the end of every 256 byte block the 22-bit parity data is stored at the end of the 2112 byte buffer ready to be written after the main 2048 byte area is written.

Listing 20-3. Page program through core write transactions

```
/*********************************************************
 Ensure the NFC write buffer is empty
*********************************************************/
P5.L  = lo(NFC_STAT);
P5.H  = hi(NFC_STAT);
_check_write_buffer_empty:
    R7.L = w[P5];
    CC   = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty;
_check_write_buffer_empty.END:
```

```
/*********************************************************
 Issue Program Command (0x80) to NAND Flash
*********************************************************/
P5.L  = lo(NFC_CMD);
P5.H  = hi(NFC_CMD);
R7.L  = 0x0080;
w[P5] = R7.L;

/*********************************************************
 In order to avoid a write buffer overflow error
 Issue 3 of the 5 address cycles to the NAND flash
 The read command and the three address cycles are enough
 to fill up the NFC write buffer.
*********************************************************/
P5.L  = lo(NFC_ADDR);
P5.H  = hi(NFC_ADDR);
R7.L  = 0x0000;
w[P5] = R7.L;
w[P5] = R7.L;
w[P5] = R7.L;

/*********************************************************
 Wait for write buffer to become empty
*********************************************************/
P5.L  = lo(NFC_STAT);
P5.H  = hi(NFC_STAT);
_check_write_buffer_empty_again:
    R7.L = w[P5];
    CC   = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty_again;
_check_write_buffer_empty_again.END:
```

```
/*********************************************************
 Issue the remaining 2 address cycles
*********************************************************/
P5.L  = lo(NFC_ADDR);
P5.H  = hi(NFC_ADDR);
R7.L  = 0x0000;
w[P5] = R7.L;
w[P5] = R7.L;

/*********************************************************
 We are now ready to start programming the page with data.
 This routine will program all 2112 bytes of the page. Four
 Bytes are programmed every iteration of the loop to make
 Most efficient use of the 4 deep write buffer
*********************************************************/
P5.L = lo(NFC_STAT);
P5.H = hi(NFC_STAT);
P4.L = lo(NFC_DATA_WR);
P4.H = hi(NFC_DATA_WR);
P3.L = lo(_Buffer);
P3.H = hi(_Buffer);
P2.L = lo(2112/4);
P2.H = hi(2112/4);

LSETUP(_write_data_begin, _write_data_end) LC0 = P2;
_write_data_begin:
    R7    = b[P3++](z);
    w[P4] = R7.L;
    R7    = b[P3++](z);
    w[P4] = R7.L;
    R7    = b[P3++](z);
    w[P4] = R7.L;
    R7    = b[P3++](z);
    w[P4] = R7.L;
```

```
    _check_writes_completed:
        R7.L = w[P5];
        CC   = bittst(R7, bitpos(WB_EMPTY));
        If !CC JUMP _check_writes_completed;
    _check_writes_completed.END:
_write_data_end:nop;

/*********************************************************
 Issue Program Confirm Command (0x10) to NAND Flash
*********************************************************/
P5.L  = lo(NFC_CMD);
P5.H  = hi(NFC_CMD);
R7.L  = 0x0010;
w[P5] = R7.L;


/*********************************************************
 Wait for the NFC not busy wakeup interrupt
*********************************************************/
P5.L = lo(NFC_IRQSTAT);
P5.H = hi(NFC_IRQSTAT);
_wait_for_ready:
    IDLE;
    R7.L = w[P5];
    CC   = bittst(R7, bitpos(NBUSYIRQ));
    IF !CC JUMP _wait_for_ready;
_wait_for_ready.END:
w[P5] = R7;
```

Listing 20-4 illustrates one method on how to perform a page read from the NAND flash through DMA.

This example assumes that only NFC wakeup interrupts are being used to bring the processor out of idle. The attached NAND flash is a large page device that requires the issuing of 5 address cycles for a read operation, the page size of the device is 2048 bytes excluding the spare area.It is assumed that the NAND flash may not be in read mode already.

Once the NAND flash is ready after the acceptance of the read confirm command DMA channel 22 is used to transfer the 2048 bytes of the main area into internal memory. This involves a single loop that is executed 8 times. Each iterations configures the DMA channel for a 256 byte read transfer and uses the DMA completion wakeup as an indication that the block read has completed.

At the end of every 256 byte block the 22-bit parity data is read from the NFC_ECC1 and NFC_ECC0 registers and stored to a second buffer in internal memory named "_CalculatedECC" before resetting the NFC ready for the next 256 byte block.

Upon completion of reading all 2048 bytes, the spare area is then read and stored at the bottom of the available 2112 byte buffer. The spare area is read through core transactions as the DMA channel can only be configured for a number of transfers that is an integer multiple of the configured NFC page size.

Listing 20-4. Page read using DMA

```
/*********************************************************
 Ensure the NFC write buffer is empty
*********************************************************/
P5.L  = lo(NFC_CTL);
P5.H  = hi(NFC_CTL);
_check_write_buffer_empty:
    R7  = w[P5 + lo(NFC_STAT - NFC_CTL)](z);
    CC = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty;
_check_write_buffer_empty.END:

/*********************************************************
 Issue Read Command (0x00) to NAND Flash
*********************************************************/
R7  = 0(x);
w[P5+ lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 In order to avoid a write buffer overflow error
 Issue 3 of the 5 address cycles to the NAND flash
 The read command and the three address cycles are enough
 to fill up the NFC write buffer.
*********************************************************/
R7 = 0(x);
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
```

```
/*********************************************************
 Wait for write buffer to become empty
*********************************************************/
_check_write_buffer_empty_again:
    R6  = w[P5 + lo(NFC_STAT-NFC_CTL)](z);
    CC = bittst(R6, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty_again;
_check_write_buffer_empty_again.END:

/*********************************************************
 Issue the remaining 2 address cycles followed by the
 Read Confirmation command
*********************************************************/
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5+lo(NFC_ADDR - NFC_CTL)] = R7;
R7  = 0x30(z);
w[P5+lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 Wait for the NFC not busy wakeup interrupt
*********************************************************/
_wait_for_ready:
    IDLE;
    R7 = w[P5+lo(NFC_IRQSTAT - NFC_CTL)](z);
    CC = bittst(R7, bitpos(NBUSYIRQ));
_wait_for_ready.END: IF !CC JUMP _wait_for_ready;

R7 = NBUSYIRQ(z);
w[P5+lo(NFC_IRQSTAT - NFC_CTL)] = R7;
```

```
/*********************************************************
 The page we wish to access is now ready to be read from
 the NAND flash. Set up the pointers to the data buffer
 and the buffer where we will store the calculated error
 correction parity data.
*********************************************************/

P4.L = lo(DMA22_CONFIG);
P4.H = hi(DMA22_CONFIG);

P3.L = lo(_CalculatedECC);
P3.H = hi(_CalculatedECC);
P2 = (2048/256)(z);

R3.L = lo(_Buffer);
R3.H = hi(_Buffer);
R4 = (256/4)(z);
R5 = 0x4(x);

LSETUP(_read_data_begin, _read_data_end) LC0 = P2;
_read_data_begin: /* Outer loop */
    /* Reset the NFC */
    R7 = ECC_RST(z);
    w[P5 + lo(NFC_RST - NFC_CTL)] = R7;
    ssync;
    _wait_for_nfc_reset_completion:
        R7 = w[P5 + lo(NFC_RST - NFC_CTL)](z);
        CC = bittst(R7, bitpos(ECC_RST));
    _wait_for_nfc_reset_completion.END:
        if CC jump _wait_for_nfc_reset_completion;

    R7 = 0x00(x);
    w[P4] = R7;
    [P4 + (DMA22_START_ADDR - DMA22_CONFIG)] = R3;
```

```
    w[P4 + lo(DMA22_X_COUNT - DMA22_CONFIG)] = R4;
    w[P4 + lo(DMA22_X_MODIFY - DMA22_CONFIG)] = R5;
    R7 = 256(z);
    R3 = R3 + R7;
    R7 = 0x8B(z);
    w[P4] = R7;
    csync;
    R7 = PG_RD_START(x);
    w[P5 + lo(NFC_PGCTL - NFC_CTL)] = R7;

    _wait_for_dma_complete:
        IDLE;
        R7 = w[P4 + lo(DMA22_IRQ_STATUS - DMA22_CONFIG)](z);
        CC = bittst(R7, bitpos(DMA_DONE));
    _wait_for_dma_complete.END: IF !CC JUMP
_wait_for_dma_complete;
    R7 = DMA_DONE(z);
    w[P4 + lo(DMA22_IRQ_STATUS - DMA22_CONFIG)] = R7;

    /* Read and store the error correction parity data */
    R7 = w[P5+ lo(NFC_ECC0 - NFC_CTL)](z);
    R6 = w[P5+ lo(NFC_ECC1 - NFC_CTL)](z);
    R0 = w[P5+ lo(NFC_COUNT - NFC_CTL)](z);
    R6 <<= 11;
    R7 = R7 | R6;
_read_data_end: [P3++] = R7;
```

```
/*********************************************************
 We now wish to read the spare area of the page that
 contains the expected error correction parity data to
 use with the newly calculated parity data
*********************************************************/
P2 = 0x40(z);
P4.L = lo(_Buffer+2048);
P4.H = hi(_Buffer+2048);

LSETUP(_read_page_spare_begin, _read_page_spare_end) LC1 = P2;
_read_page_spare_begin:
    w[P5+ lo(NFC_DATA_RD - NFC_CTL)] = R7;

    _wait_for_spare_data_ready:
            IDLE;
            R7 = w[P5+ lo(NFC_IRQSTAT - NFC_CTL)];
            CC = bittst(R7, bitpos(RD_RDY));
            IF !CC JUMP _wait_for_spare_data_ready;
        _wait_for_spare_data_ready.END:


/*********************************************************
        The byte is now available in the NFC_READ register.
        We need to read the byte which results in the read
        transaction then being removed from the write buffer.
        We need to ensure that this transaction completes before
        clearing the IRQ
 *********************************************************/
        R7 = RD_RDY(z);
        R6 = w[P5+ lo(NFC_READ - NFC_CTL)](z);
        ssync;
        w[P5+ lo(NFC_IRQSTAT - NFC_CTL)] = R7;
    _read_page_spare_end: b[P4++] = R6;
```

Listing 20-5 illustrates one method on how to perform a page program to the NAND flash through DMA. This example assumes that only NFC wakeup interrupts are being used to bring the processor out of idle.

The attached NAND flash is a large page device that requires the issuing of 5 address cycles for a read operation, the page size of the device is 2048 bytes excluding the spare area. It is assumed that the NAND flash may not be in program mode already.

Once the page program command and address cycles have been issued to the NAND flash the data cycles are then initiated through DMA channel 22 to transfer the 2048 bytes of the main area.

This example works differently from the read example through DMA in that multiple DMA sequences are not configured. For a page write transaction a full 2048 byte DMA can be configured. The PG_WR_START bit in the NFC_PGCTL register is then used to start each smaller DMA sequence. As the NFC is assumed to be configured for 256 byte page size in the NFC_CTL register, each issue of the page write start will only allow the DMA to transfer 256 bytes. This is performed within the single loop executed 8 times to transfer the 2048 byte page.

At the end of every 256 byte block the 22-bit parity data is read from the NFC_ECC1 and NFC_ECC0 registers and stored at the end of the 2112 byte buffer.

Upon completion of the 2048 byte DMA, the spare area is then written through core write transaction before issuing a page program confirmation command.

Listing 20-5. Page program using DMA

```
/*********************************************************
 Ensure the NFC write buffer is empty
*********************************************************/
P5.L  = lo(NFC_CTL);
P5.H  = hi(NFC_CTL);
_check_write_buffer_empty:
    R7 = w[P5 + lo(NFC_STAT - NFC_CTL)](z);
    CC   = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty;
_check_write_buffer_empty.END:

/*********************************************************
 Issue Program Command (0x80) to NAND Flash
*********************************************************/
R7 = 0x80(z);
w[P5 + lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 In order to avoid a write buffer overflow error
 Issue 3 of the 5 address cycles to the NAND flash
 The read command and the three address cycles are enough
 to fill up the NFC write buffer.
*********************************************************/
R7  = 0x00(x);
w[P5 + lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5 + lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5 + lo(NFC_ADDR - NFC_CTL)] = R7;
```

```
/*********************************************************
 Wait for write buffer to become empty
*********************************************************/
_check_write_buffer_empty_again:
    R6 = w[P5 + lo(NFC_STAT - NFC_CTL)](z);
    CC   = bittst(R6, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty_again;
_check_write_buffer_empty_again.END:

/*********************************************************
 Issue the remaining 2 address cycles
*********************************************************/
w[P5 + lo(NFC_ADDR - NFC_CTL)] = R7;
w[P5 + lo(NFC_ADDR - NFC_CTL)] = R7;

/*********************************************************
 Wait for write buffer to become empty
*********************************************************/
_check_write_buffer_empty_yet_again:
    R7 = w[P5 + lo(NFC_STAT - NFC_CTL)](z);
    CC   = bittst(R7, bitpos(WB_EMPTY));
    If !CC JUMP _check_write_buffer_empty_yet_again;
_check_write_buffer_empty_yet_again.END:

/*********************************************************
 We are now ready to start programming the page with data.
 This routine will program all 2048bytes of the page
*********************************************************/
P4.L = lo(DMA22_CONFIG);
P4.H = hi(DMA22_CONFIG);
P2.L = lo(_Buffer+2048);
P2.H = hi(_Buffer+2048);
R7 = 0(x);
w[P4] = R7;
```

```
P1 = (2048/256)(z);

R3.L = lo(_Buffer);
R3.H = hi(_Buffer);
R4 = (2048/4)(z);
R5 = 0x4(x);

[P4 + (DMA22_START_ADDR - DMA22_CONFIG)] = R3;
w[P4 + lo(DMA22_X_COUNT - DMA22_CONFIG)] = R4;
w[P4 + lo(DMA22_X_MODIFY - DMA22_CONFIG)] = R5;
R7 = 0x89(z);
w[P4] = R7;

LSETUP(_write_data_begin, _write_data_end) LC0 = P1;
_write_data_begin:
    P1 = (256/4)(z);
    /* Reset the NFC */
    R7 = ECC_RST(z);
    w[P5 + lo(NFC_RST - NFC_CTL)] = R7;
    ssync;
    _wait_for_nfc_reset_completion:
        R7 = w[P5 + lo(NFC_RST - NFC_CTL)](z);
        CC = bittst(R7, bitpos(ECC_RST));
    _wait_for_nfc_reset_completion.END:
        if CC jump _wait_for_nfc_reset_completion;

    R7 = PG_WR_START(x);
    w[P5 + lo(NFC_PGCTL - NFC_CTL)] = R7;
```

```
    _wait_for_page_write_complete:
        IDLE;
        R7 = w[P5 + lo(NFC_IRQSTAT - NFC_CTL)](z);
        CC = bittst(R7, bitpos(WR_DONE));
    _wait_for_page_write_complete.END: IF !CC JUMP
_wait_for_page_write_complete;
    R7 = WR_DONE(z);
    w[P5 + lo(NFC_IRQSTAT - NFC_CTL)] = R7;

    /* Read and store the error correction parity data */
    R7 = w[P5+ lo(NFC_ECC0 - NFC_CTL)](z);
    R6 = w[P5+ lo(NFC_ECC1 - NFC_CTL)](z);
    R6 <<= 11;
    R7 = R7 | R6;
    [P2++] = R7;
_write_data_end: P2+=4;

R7 = DMA_DONE(z);
w[P4 + (DMA22_IRQ_STATUS - DMA22_CONFIG)] = R7;

/*********************************************************
 We are now ready to start writing the spare area of the
 page now we have collected all the parity data
*********************************************************/
P1 = (64/4)(z);
P4.L = lo(_Buffer+2048);
P4.H = hi(_Buffer+2048);

LSETUP(_write_data_spare_begin, _write_data_spare_end) LC0 = P1;
_write_data_spare_begin:
    R7 = b[P4++](z);
    R6 = b[P4++](z);
    R5 = b[P4++](z);
    R4 = b[P4++](z);
```

```
    w[P5 + lo(NFC_DATA_WR - NFC_CTL)] = R7;
    w[P5 + lo(NFC_DATA_WR - NFC_CTL)] = R6;
    w[P5 + lo(NFC_DATA_WR - NFC_CTL)] = R5;
    w[P5 + lo(NFC_DATA_WR - NFC_CTL)] = R4;

    _check_writes_spare_completed:
        R7 = w[P5 + lo(NFC_STAT - NFC_CTL)];
        CC   = bittst(R7, bitpos(WB_EMPTY));
        If !CC JUMP _check_writes_spare_completed;
_write_data_spare_end:nop;

/*********************************************************
 Issue Program Confirm Command (0x10) to NAND Flash
*********************************************************/
R7  = 0x10(z);
w[P5+ lo(NFC_CMD - NFC_CTL)] = R7;

/*********************************************************
 Wait for the NFC not busy wakeup interrupt
*********************************************************/
_wait_for_ready:
    IDLE;
    R7= w[P5 + lo(NFC_IRQSTAT - NFC_CTL)](z);
    CC   = bittst(R7, bitpos(NBUSYIRQ));
    IF !CC JUMP _wait_for_ready;
_wait_for_ready.END:
R7 = NBUSYIRQ(z);
w[P5+lo(NFC_IRQSTAT - NFC_CTL)] = R7;
```

21 ATAPI INTERFACE

This chapter describes the processor's advanced technology attachment packet interface (ATAPI). This interface is an ATA/ATAPI-6 compliant host implementation. The ATA interface, also known as the IDE (Integrated Drive Electronics) interface, provides a simple interface to low-cost non-volatile memories like hard-disk drives, DVD players, CDROM players/writers, and compact flash and PC-card devices. The ATAPI interface supports all ATA hardware protocol transfers and the complete set of 80 ATAPI commands.

This chapter includes the following sections:

- "Interface Overview" on page 21-1
- "Description of Operation" on page 21-4
- "Functional Description" on page 21-18
- "Programming Model" on page 21-40
- "ATAPI Registers" on page 21-46
- "ATAPI Standards Reference" on page 21-73

Interface Overview

The ATAPI interface supports all ATA hardware protocol transfers and the complete set of 80 ATAPI commands.

The ATAPI includes these features:

- ATA/ATAPI-6 compliant core supports:
  - PIO modes 0, 1, 2, 3, 4
  - Multiword DMA modes 0, 1, 2
  - Ultra DMA modes 0, 1, 2, 3, 4, 5 (up to UDMA 100)
- Programmable timing parameters to support ATA interface timing at any processor clock frequency
- Interface to compact flash (CF) configured in True-IDE mode

Figure 21-1 shows a block diagram of the ATAPI block. The ATAPI host interfaces to the rest of the system through the PAB and DAB buses. The PAB bus is used for programming the control and status registers. The DAB buses are used for transmitting and receiving ATAPI packets

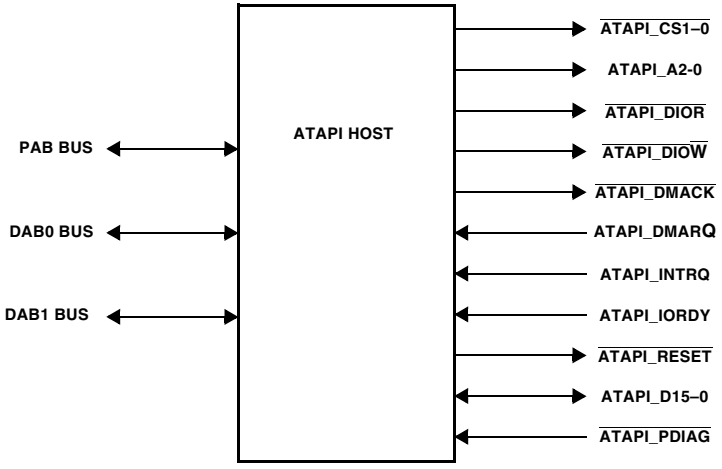


Figure 21-1. ATAPI Block Diagram

The ATAPI shares its pins with other peripherals on chip. For more information see Chapter 9, “General-Purpose Ports”.

Table 21-1 lists the signal pins for the ATAPI block.

Table 21-1. ATAPI Signals Summary

Signal	Dir	Description
ATAPI_CS1-0	O	Chip select signals from the host used to select the command block or control block registers. When ATAPI_DMACK is asserted, ATAPI_CS1-0 is negated and transfers are 16 bits wide.
ATAPI_A2-0	O	This is a 3-bit binary code address asserted by the host to access the register or data port in the device.
ATAPI_DIOR ATAPI_HDMARDY ATAPI_HSTROBE	O	ATAPI_DIOR is the strobe signal asserted by the host to read the device register or data port.
ATAPI_DIOW ATAPI_STOP	O	ATAPI_DIOW is the strobe signal asserted by the host to write the device register or data port
ATAPI_DMACK	O	This signal is used in response to ATAPI_DMARQ to initiate DMA transfers
ATAPI_DMARQ	I	Asserted by the device during DMA transfers and held until acknowledged by the host via ATAPI_DMACK. The host can pause the DMA transfer by deasserting ATAPI_DMARQ. At the same time, ATAPI_DMACK can be continuously asserted if more DMA data is available from the host.
ATAPI_INTRQ	I	Used by the selected device to interrupt the host when interrupt is pending.
ATAPI_IORDY ATAPI_DDMARDY ATAPI_DSTROBE	I	The device can create wait state when it is not ready to respond for any host register access (read or write).
ATAPI_RESET	O	Used by the host as a hard reset to reset the devices connected on the ATAPI bus.
ATAPI_D15–0	I/O	Data Bus for ATAPI interface
ATAPI_PDIAG	I	Used to determine if an 80-pin cable is connected to the host.

Description of Operation

The complete set of ATAPI commands (80) can be categorized into the following transfer types:

- Programmable IO
- Device register IO
- Multi-word DMA mode

Host PIO/Register Transfers

A write or a read from an address in the 0x00 to 0x0F range to the ATAPI_DEV_ADDR register with PIO_START set initiates a PIO or a Register transfer. For address 0x00, PIO data port transfers are initiated; whereas for all other address values, a register access transfer is initiated.

The sequence of operation for any register transfer is as follows:

- Program the PIO and register timing registers based on the mode supported by device (decoded by IDENTIFY DEVICE COMMAND)
- For device register write
    - Program the ATAPI_DEV_TXBUF register with write data (to be written into the device).
    - Program the ATAPI_DEV_ADDR register with address of the device register (0x01 to 0x0F).
    - Set the appropriate interrupt mask (PIO_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
    - Program the ATAPI_CONTROL register with XFER_DIR set to write (1) and PIO_START set to 1.

- Wait for the interrupt to indicate the end of the transfer.
- Alternately, the software can poll the PIO_XFER_ON bit in ATAPI_STATUS register to wait for the completion of the transfer.

- For device register read
  - Program the ATAPI_DEV_ADDR register with address of device register (0x01 to 0x0F)
  - Set the appropriate interrupt mask (PIO_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
  - Program the ATAPI_CONTROL register with XFER_DIR bit set to read (0) and PIO_START set to 1.
  - Wait for the interrupt to indicate the end of the read operation.
  - Alternatively, the software can poll the PIO_XFER_ON bit in ATAPI_STATUS register to wait for the completion of the transfer.
  - Read the ATAPI_DEV_RXBUF register to obtain the device register value.

PIO Data-Out Transfers (Device Write)

This class includes the following commands:

- CFA WRITE MULTIPLE WITHOUT ERASE
- CFA WRITE SECTORS WITHOUT ERASE
- DOWNLOAD MICROCODE
- SECURITY DISABLE PASSWORD

- SECURITY ERASE UNIT
- SECURITY SET PASSWORD
- SECURITY UNLOCK
- WRITE BUFFER
- WRITE MULTIPLE
- WRITE SECTOR(S)

Execution of this class of command includes transfer of one or more blocks of data from host to device (See Figure 21-2).

A basic PIO data-out command protocol involves the following sequence:

- Program the ATAPI_XFER_LEN register with the number of ATA words (1 sector = 256 ATA words) to be transfer. The following sequence is required on interrupt: (1) set ATAPI_DEV_TXBUF with the next word to transfer; (2) reset PIO_START to 1. This is similar with the PIO read sequence except that the ATAPI_DEV_RXBUF is read after each interrupt.
- Program the ATAPI_DEV_ADDR register with device PIO data port address (0x00).
- Program the ATAPI_CONTROL register with XFER_DIR bit to write (1).
- Set the ATAPI_DEV_TXBUF register with the first word to transfer.
- Enable the appropriate interrupt (PIO_DONE_INT) in the ATAPI_INT_MASK register.
- Set the ATAPI_DEV_TXBUF register with the first word to transfer.
- Set PIO_START to 1 to start the PIO transfer.

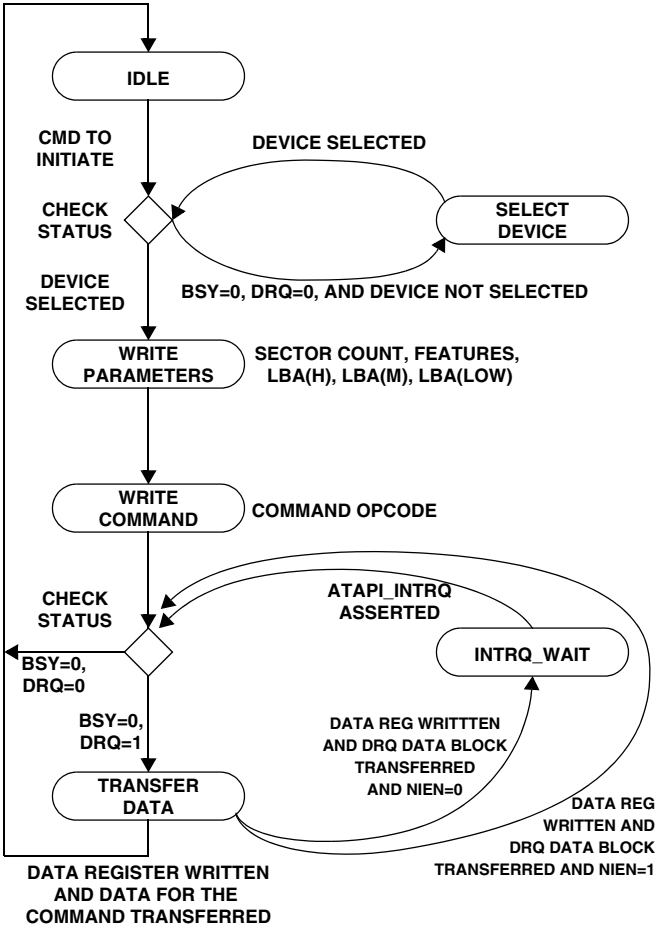


Figure 21-2. PIO Data-Out Protocol State Machine (Device Write)

- Wait for the interrupt to indicate the completion of the PIO transfer.
- Alternatively, the software can poll the `PIO_XFER_ON` bit in `ATAPI_STATUS` register to wait for the completion of the transfer.

PIO Data-In Transfers (Device Read)

This class includes:

- CFA TRANSLATE SECTOR
- IDENTIFY DEVICE
- IDENTIFY PACKET DEVICE
- READ BUFFER
- READ MULTIPLE
- READ SECTOR (S)
- SMART READ DATA

Execution of this class of command includes transfer of one or more blocks of data from device to the host (See Figure 21-3).

A basic PIO data-in command protocol transfer involves the following sequence:

- Program the ATAPI_XFER_LEN register with number of ATA words (1 sector = 256 ATA words) that need to be transferred.
- Program the ATAPI_DEV_ADDR register with device PIO data port address (0x00).
- Program the ATAPI_CONTROL register with XFER_DIR bit set to read (0).
- Enable the appropriate interrupt (PIO_DONE_INT) in the ATAPI_INT_MASK register.
- Set the ATAPI_DEV_RXBUF register with the first word to transfer.
- Set PIO_START to 1 to start PIO transfer.

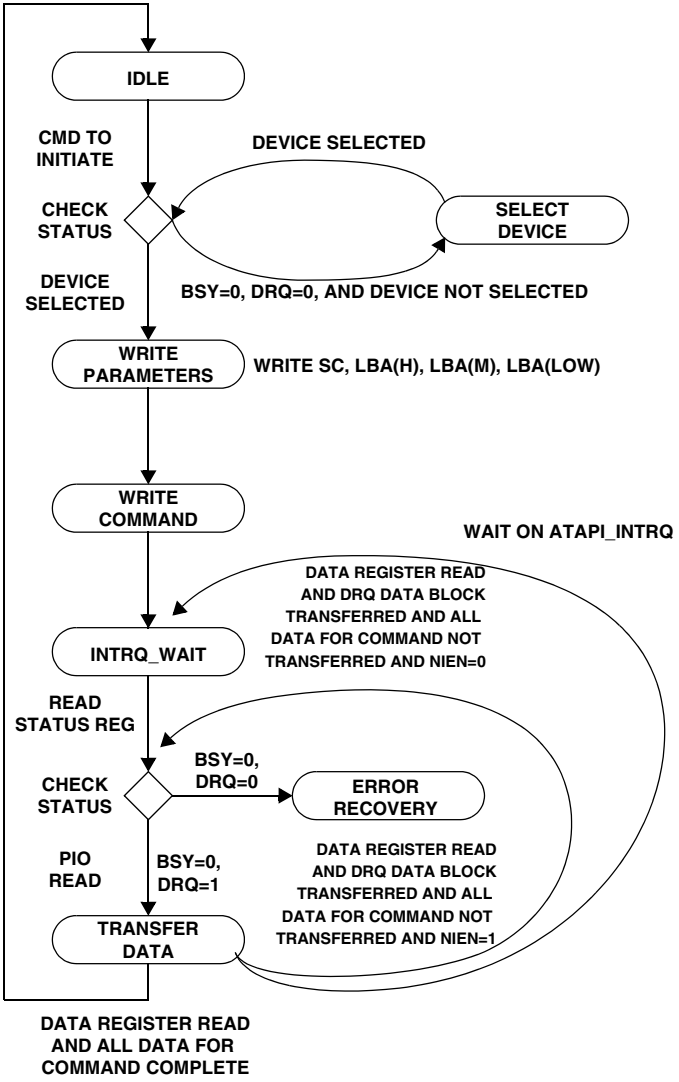


Figure 21-3. PIO Data-In State Machine (Device Read)

- Wait for the interrupt to indicate the completion of the PIO transfer.
- Alternatively, the software can poll the PIO_XFER_ON bit in ATAPI_STATUS register to wait for the completion of the transfer.

Host Multiword DMA Transfers

This class includes:

- READ DMA
- WRITE DMA

Execution of this class of command includes the transfer of one or more blocks of data from the host to device or device to host using multi DMA command protocol. The host should initialize the DMA channel prior to transferring data by executing SET_FEATURE command.

A single interrupt is issued by the device at the completion of successful transfer of all data required by the command or when the transfer is aborted due to error, whereas in case of PIO command protocol transfers, interrupt is issued after the end of every DRQ block of data transfer.

Each operation involves the following sequence:

- Program the multiword DMA Timing Registers (Based on the mode detected by IDENTIFY DEVICE command).
- For a block of DMA data write transfer.
    - Program ATAPI_XFER_LEN register with the number of ATA words to be transferred.
    - Program ATAPI_CONTROL register with XFER_DIR bit set to write (1).

- Set the appropriate interrupt mask (MULTI_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
- Set MULTI_START bit to 1.
- Wait for the interrupt to indicate the completion of the transfer.

- For a block of DMA data read transfer.
  - Program ATAPI_XFER_LEN register with number of ATA words to be transferred.
  - Program ATAPI_CONTROL register with XFER_DIR bit set to read (0).
  - Set the appropriate interrupt mask (MULTI_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
  - Set MULTI_START bit to 1.
  - Wait for the interrupt to indicate the end of the transfer.

For second device: Reprogram the DMA Timing Registers, select the second device by writing in to device register and start DMA read/write operations.

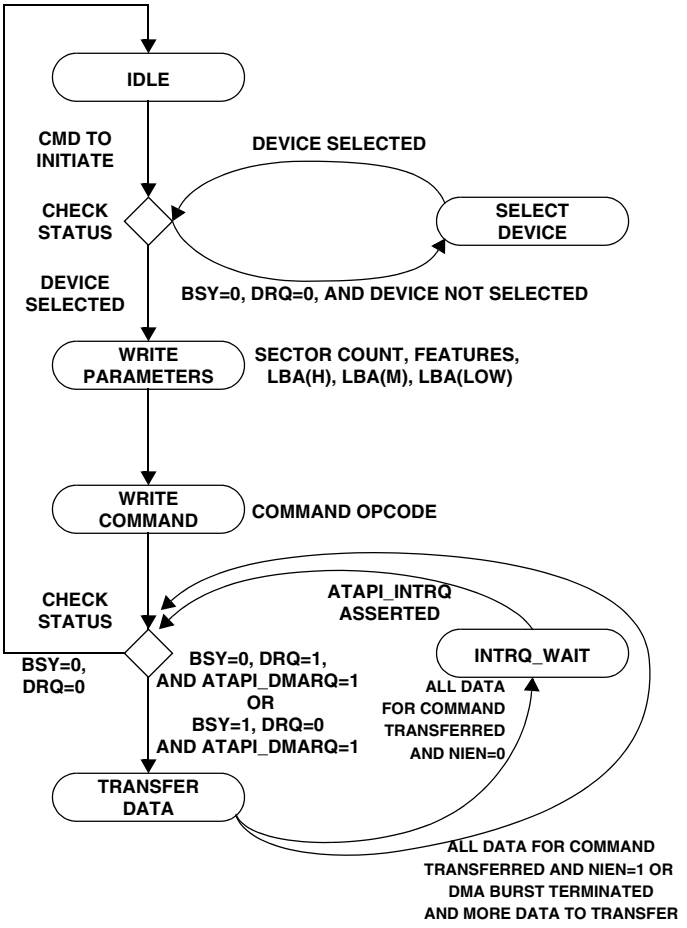


Figure 21-4. Host DMA State Machine

Host Pausing the Multi-DMA Transfer

The ATAPI host pauses any current multi-DMA transfer when data may not be immediately available from the system. This is accomplished by not generating further $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ pulses and at same time keeping the $\overline{\text{ATAPI\_DMACK}}$ asserted so that the device does not go for termination. Once the host is ready, it starts the remaining transfers by generating the $\overline{\text{ATAPI\_DIOW}}$/$\overline{\text{ATAPI\_DIOR}}$ pulses.

Host Terminating the Multi DMA Transfer

The host can terminate the current multi-DMA data transfer (based on detecting that the current on-going transfer is erroneous) before data transfer is completed by setting the ATAPI_TERMINATE register bit. The ATAPI host initiates the termination by negating $\overline{\text{ATAPI\_DMACK}}$ within t_J after an $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ pulse. If the device is able to continue the transfer of data, the device may leave ATAPI_DMARQ asserted and wait for the host to reassert $\overline{\text{ATAPI\_DMACK}}$ or may negate ATAPI_DMARQ at any time after detecting that $\overline{\text{ATAPI\_DMACK}}$ is negated.

The ATAPI host needs to check the ATAPI_DMARQ line status in the ATAPI_LINE_STATUS register before issuing any new command. If it detects that the device is still waiting for the $\overline{\text{ATAPI\_DMACK}}$ to assert for the current transfer, the ATAPI host should either soft reset or hard reset the device.

Device Pausing the Multi-DMA Transfer

To pause the multi-DMA burst, the device negates the ATAPI_DMARQ line within t_L after assertion of the current $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ pulse. In multiword DMA mode, the device uses the same mechanism (negating ATAPI_DMARQ line) to indicate a pause or a termination. The ATAPI host by default gives control back to the firmware by generating a MULTI_TERM_INT interrupt. It is the responsibility of the firmware to determine if the transfer is paused or terminated.

The number of pause interrupts to service can be reduced considerably by setting the END_ON_TERM bit in the ATA_CONTROL register to 1. This allows the ATAPI host to go into a pause state until the device is ready to transfer the data.

If the Host Automatic Pause Handling is not used, the user needs to restart the multiword DMA transfer by setting MULTI_START to 1 to continue the transfer after determining that the device has paused.

When using the END_ON_TERM bit, it is mandatory to request and interrupt from the device by setting the nIEN bit. This enables catching any error conditions that might occur during a multiword transfer.

Once the device asserts back the ATAPI_DMARQ, the host restarts generating $\overline{\text{ATAPI\_DIOW}}$/$\overline{\text{ATAPI\_DIOR}}$ pulses and completes the transfer of the remaining blocks of data.

Device Terminating the Multi-DMA Transfer

To terminate the multi-DMA burst, the device negates the ATAPI_DMARQ within t_L after the assertion of the current $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ pulse. The last word for the burst is then transferred by the negation of the current $\overline{\text{ATAPI\_DIOR}}$ or $\overline{\text{ATAPI\_DIOW}}$ pulse. If all the data for the command has not been transferred, the device re-asserts ATAPI_DMARQ again at any later time to resume multi-DMA operation. Under this condition, the ATAPI host goes into pause state waiting for the ATAPI_DMARQ line to be asserted. The ATAPI host can wait for a certain period and terminate the DMA transfer by setting ATAPI_TERMINATE register bit or by a soft reset of the ATAPI state machine by setting the SOFT_RESET register bit. After setting these bits, the host should check back to see if the ATAPI_TERMINATE register bit got cleared.

Host Ultra DMA Command Protocol Transfers

The Ultra DMA transfers are similar to DMA transfers with respect to the host software. It is only the hardware timing specification and signal-level handshaking protocol within the device that is different.

The sequence of operation for Ultra DMA transfers is:

- Program the Ultra DMA timing registers with the mode supported by the device (decoded after the INDENTIFY DEVICE command)
- For Ultra DMA data-out (Device Writes)
    - Program the ATAPI_XFER_LEN register with the number of ATA words to be transferred.
    - Set the appropriate interrupt mask (ULTRA_OUT_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
    - Program the ATAPI_CONTROL register with the XFER_DIR bit set to write (1) and ULTRA_START set to 1.
    - Wait for the interrupt to indicate the end of the transfer.
- For Ultra DMA data-in (Device Reads)
    - Program the ATAPI_XFER_LEN register with the number of ATA words to be transferred.
    - Set the appropriate interrupt mask (ULTRA_OUT_DONE_INT) in the ATAPI_INT_MASK register to enable interrupts.
    - Program the ATAPI_CONTROL register with the XFER_DIR bit set to read (0) and ULTRA_START set to 1.
    - Wait for the interrupt to indicate the end of the transfer.

Host Pausing the Ultra DMA Data-In Transfer

The ATAPI host pauses any current Ultra DMA transfer when data may not be immediately available from the system. The ATAPI host pauses the Ultra DMA data-in transfers by negating $\overline{\texttt{ATAPI\_HDMARDY}}$. The device stops generating the ATAPI_DSTROBE edges with in t_{RFS} -(75ns mode0 to 50ns mode2) of the host negating $\overline{\texttt{ATAPI\_HDMARDY}}$. After this host waits for another zero, one, two, or three additional data words and then releases the ATAPI_DD data pins by three-stating it. The additional data words are a result of cable round trip delay and t_{RFS} timing for the device.

According to the specification, the host should never pause an Ultra DMA burst until at least one data word of an Ultra DMA burst is transferred.

Host Terminating the Ultra DMA Data-In Transfer

The host terminates the current Ultra DMA data-in transfer (based on detecting that the current on going transfer is erroneous) before the transfer is completed by setting the ATAPI_TERMINATE register bit. The ATAPI host initiates Ultra DMA burst termination by negating $\overline{\texttt{ATAPI\_HDMARDY}}$ and following the sequence as given in Specification Sec 9.13.4.2 of ATAPI 4.0. The turn around time for complete termination can vary depending on the device behavior, as the host should be able to receive zero, one, and two additional data words after negating $\overline{\texttt{ATAPI\_HDMARDY}}$.

According to the specification, the host should never initiate Ultra DMA burst termination until at least one data word of Ultra DMA burst is transferred.

Device Pausing the Ultra DMA Data-In Transfer

The device can pause the Ultra DMA data-in burst by not generating additional ATAPI_DSTROBE edges.

Device Terminating the Ultra DMA Data-In Transfer

The device can terminate the Ultra DMA data-in burst sequence before the data for the current command is complete. This causes the ULTRA_IN_TERMINATED bit to set in the ATAPI_INT_STATUS register. This event is to be transferred to higher layer software, which can be validated by reading the device error register.

Host Pausing Ultra DMA Data-Out Transfer

The ATAPI host pauses the Ultra DMA data-out transfers by not generating ATAPI_HSTROBE edges and three-stating the ATAPI_Dx data pins in response to PIN release for higher priority peripherals. At same time, the ATAPI host keeps the $\overline{\text{ATAPI\_DMACK}}$ asserted and the HSTOP de-asserted. This should not make the device start a termination sequence, as the $\overline{\text{ATAPI\_DMACK}}$ is still kept asserted.

Host Terminating Ultra DMA Data-Out Transfer

The host terminates the current Ultra DMA data-out transfer before the transfer is completed by setting the TERMINATE bit in the ATAPI_CONTROL register. The ATAPI host starts the termination sequence by not generating ATAPI_HSTROBE edges, followed by asserting HSTOP, followed by de-asserting $\overline{\text{ATAPI\_DMACK}}$.

Device Pausing the Ultra DMA Data-Out Transfer

The device can pause an Ultra DMA data-out burst by negating $\overline{\text{ATAPI\_DDMARDY}}$. The ATAPI host enters into a pause state and waits until the device asserts $\overline{\text{ATAPI\_DDMARDY}}$.

Device Terminating the Ultra DMA Data-Out Transfer

The device terminates the Ultra DMA data-out sequence by negating ATAPI_DDMARDY before complete data is transferred and then negating ATAPI_DMARQ after t_{RP}. The ATAPI host enters the pause state once it sees the ATAPI_DDMARDY getting de-asserted. During pause state, if it sees the ATAPI_DMARQ getting de-asserted, it goes into the termination sequence. This results in the ULTRA_OUT_TERMINATED bit getting set in ATAPI_INT_STATUS register.

Functional Description

The following sections describe the function of the various protocols and functions in the ATAPI controller. For more detailed information on exact timing parameters, refer to the ATA/ATAPI-6 Specification and *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Power-on and Hardware Reset Protocol

The ATAPI host can use the DEV_RST bit in the ATAPI_CONTROL register to drive the ATAPI_RESET pin of the device. When the ATAPI_RESET signal is asserted, the connected devices execute the hardware reset protocol. The host should respond as described below:

1. Assert ATAPI_RESET for at least 25 µs by writing a value of 1 to the DEV_RST bit (can use one of the system timers).
2. Negate ATAPI_RESET by writing a 0 to the DEV_RST bit and wait at least 2ms.
3. Read the device status register or the alternate status register.
4. Wait for the busy flag (BSY) to be cleared.

5. Perform an IDENTIFY DEVICE or IDENTIFY PACKET DEVICE command for each connected device.
6. Read the device parameters from each connected device.
7. Program the ATAPI host's timing registers depending on the data read from the device(s).

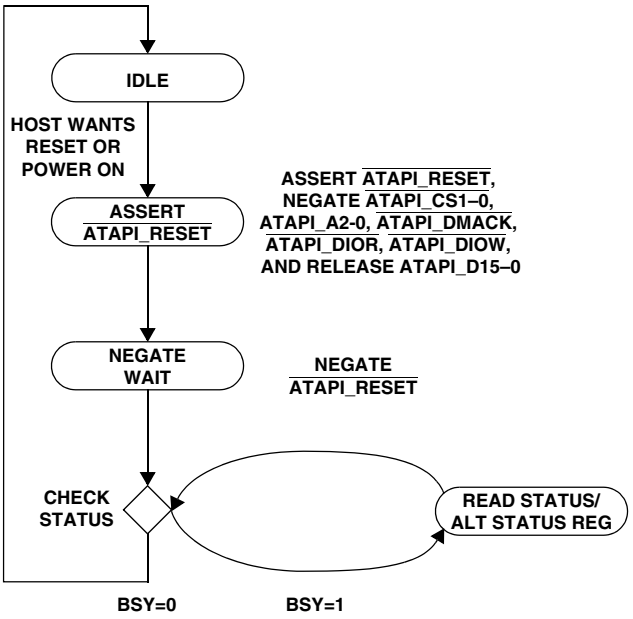


Figure 21-5. Power-On and Hardware Reset Protocol

Device Selection Protocol

Before issuing any command to a device except the DEVICE RESET command, the host should ensure that the selected device is no longer busy, select the desired device, and insure that it is ready to accept a command. Figure 21-6 below describes the protocol for device selection.

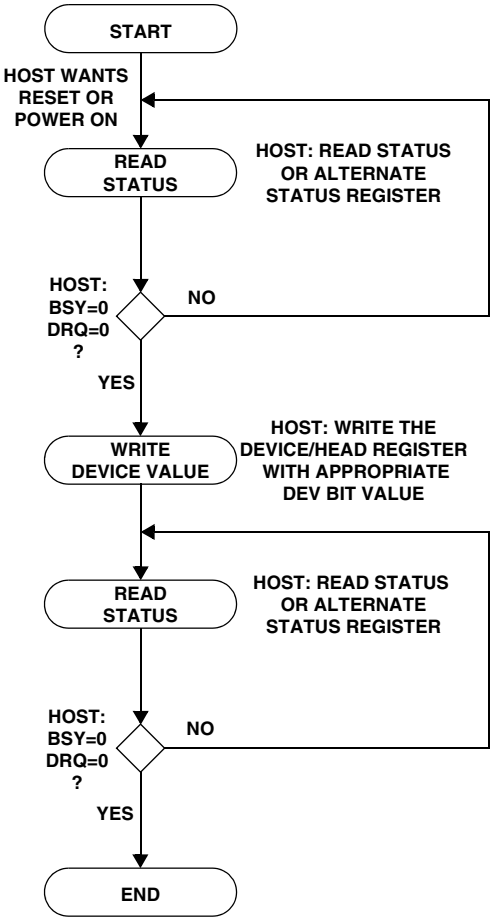


Figure 21-6. Device Selection Protocol

Programmed I/O (PIO)

A write to or a read from an address in the 0x00 to 0x0F range initiates a PIO write or read transfer respectively. The PIO registers of the device are mapped into this range. When the core detects a read or write access in this address range, with the PIO_START bit set, it executes a PIO Transfer cycle or Device IO register transfer cycle as shown in Figure 21-7. The following notes apply to Figure 21-7.

- ADDR consists of signals: $\overline{\text{ATAPI\_CS1-0}}$ and ATAPI_A2-0.
- DATA consists of ATAPI_D15-0 for all devices except devices implementing the CFA feature set when 8-bit transfers are enabled. In that case, DATA consists of ATAPI_D7-0.
- The negation of ATAPI_IORDY by the device is used to extend the PIO cycle. The determination of whether the cycle is to be extended is made by the host after t_A from the assertion of $\overline{\text{ATAPI\_DIOR}}$ or $\overline{\text{ATAPI\_DIOW}}$. The assertion and negation of ATAPI_IORDY are described in the following three cases:
  - Device never negates ATAPI_IORDY, devices keep ATAPI_IORDY released: no wait is generated.
  - Device negates ATAPI_IORDY before t_A, but causes ATAPI_IORDY to be asserted before t_A. ATAPI_IORDY is released prior to negation and may be asserted for no more than 5 ns before release: no wait is generated.
  - Device negates ATAPI_IORDY before t_A. ATAPI_IORDY is released prior to negation and may be asserted for no more than 5 ns before release: wait is generated. The cycle completes after ATAPI_IORDY is reasserted. For cycles where a wait is generated and $\overline{\text{ATAPI\_DIOR}}$ is asserted, the device places read data on ATAPI_D7-0 for the t_{RD} before asserting ATAPI_IORDY.

- $\overline{\texttt{ATAPI\_DMACK}}$ is negated during a PIO data transfer.

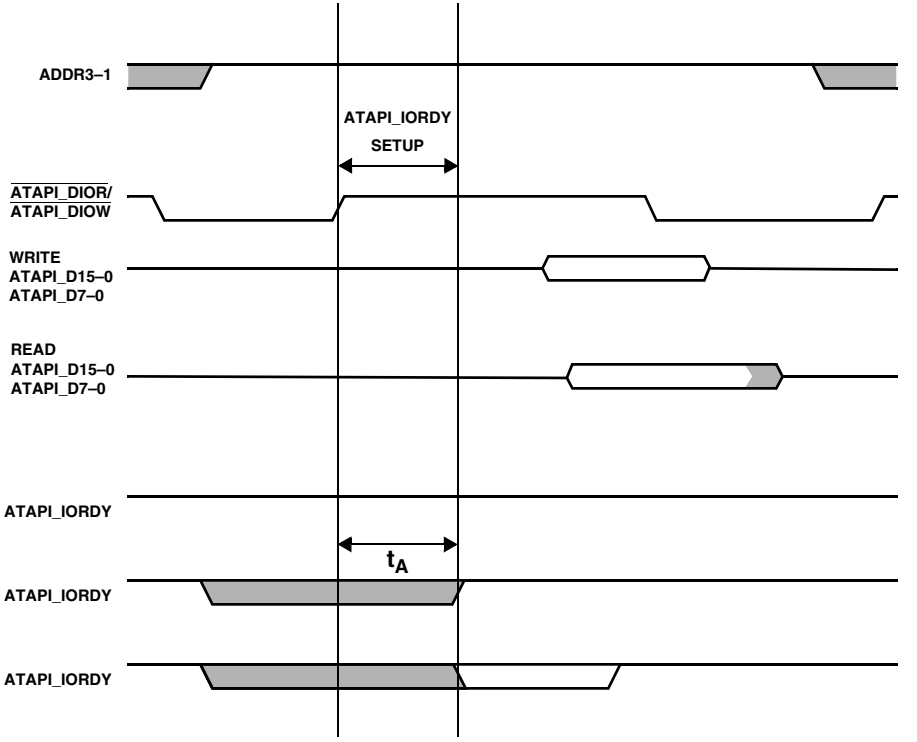


Figure 21-7. PIO Data Transfer to/from the Device Timing Diagram

Host Multi DMA Block Implementation

The ATAPI device initiates a multi DMA transfer by asserting the ATAPI_DMARQ line. It does so in response to READ DMA, WRITE DMA, READ DMA QUEUED, WRITE DMA QUEUED and PACKET commands. When the multiword DMA timing registers are programmed, and the MULTI_START bit is set, the host responds to the assertion of ATAPI_DMARQ by starting a multi DMA transfer cycle as shown in

Figure 21-8. Either the device or the host can terminate the transfer cycle. The device terminates the cycle by negating ATAPI_DMARQ; the host terminates the cycle by negating ATAPI_DMACK.

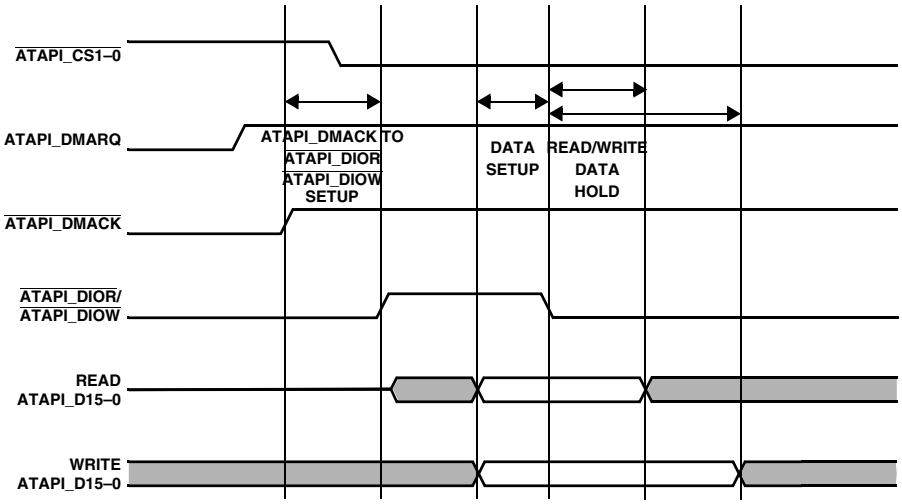


Figure 21-8. Initiating a Multiword DMA Burst

The direction of the data transfer is controlled by the command issued to the ATAPI devices and the XFER_DIR bit in the ATAPI_CONTROL register. When set (1), the core's response is a multi-DMA write cycle. When cleared (0), the core's response is a multi-DMA read cycle.

Setting the XFER_DIR bit to write (1) while a READ DMA (QUEUED) command is issued or setting the XFER_DIR bit to read (0) while a WRITE DMA (QUEUED) command is issued can lead to unpredictable results and a deadlock condition (see Figure 21-9).

Table 21-2. Multiword DMA Transfer Timing Table

Multiword DMA Timing Parameters	
t_0	Cycle time[1]
t_D	$\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ asserted pulse width[1]
t_E	$\overline{\text{ATAPI\_DIOR}}$ data access
t_F	$\overline{\text{ATAPI\_DIOR}}$ data hold
t_G	$\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ data setup
t_H	$\overline{\text{ATAPI\_DIOW}}$ data hold
t_I	$\overline{\text{ATAPI\_DMACK}}$ to $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ setup
t_J	$\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$ to $\overline{\text{ATAPI\_DMACK}}$ hold
t_{KR}	$\overline{\text{ATAPI\_DIOR}}$ negated pulse width[1]
t_{KW}	$\overline{\text{ATAPI\_DIOW}}$ negated pulse width[1]
t_{LR}	$\overline{\text{ATAPI\_DIOR}}$ to ATAPI_DMARQ delay
t_{LW}	$\overline{\text{ATAPI\_DIOW}}$ to ATAPI_DMARQ delay
t_M	$\overline{\text{ATAPI\_CS1-0}}$ valid to $\overline{\text{ATAPI\_DIOR}}$/$\overline{\text{ATAPI\_DIOW}}$
t_N	$\overline{\text{ATAPI\_CS1-0}}$ hold
t_Z	$\overline{\text{ATAPI\_DMACK}}$ to read data released

1 For exact timing information, refer to the ATA/ATAPI-6 Specification and *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

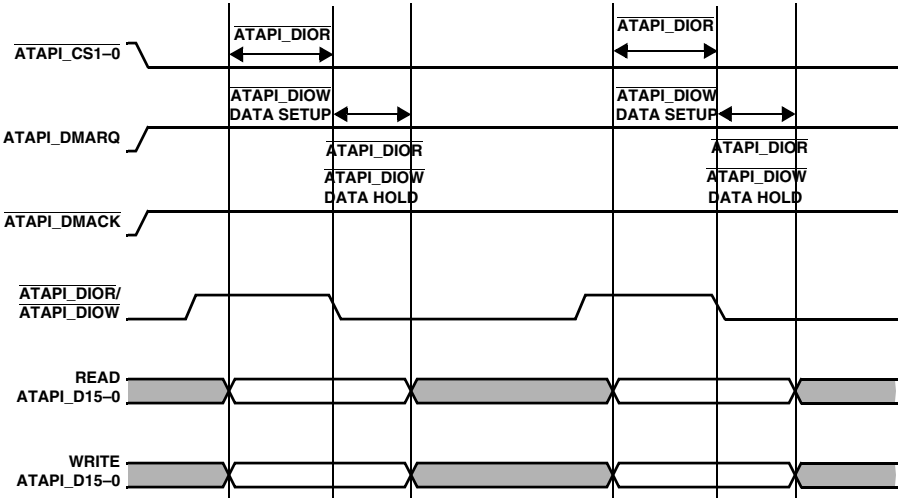


Figure 21-9. Sustaining a Multiword DMA Data Burst

To terminate the data burst, the device negates the ATAPI_DMARQ within t_L of the assertion of the current $\overline{\text{ATAPI\_DIOR}}$ or $\overline{\text{ATAPI\_DIOW}}$ pulse. The last data word for the burst is then transferred by the negation of the current $\overline{\text{ATAPI\_DIOR}}$ or $\overline{\text{ATAPI\_DIOW}}$ pulse. If all data for the command has not been transferred, the device re-asserts the ATAPI_DMARQ again at a later time to resume the DMA operation as shown in Figure 21-10.

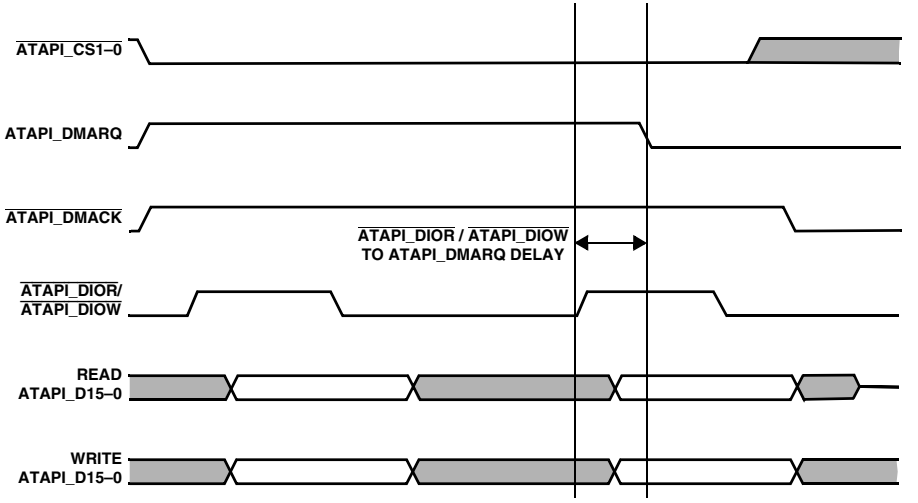


Figure 21-10. Device Terminating a Multiword DMA Burst

To terminate the transmission of a data burst, the host negates ATAPI_DMACK within t_J after a ATAPI_DIOR or ATAPI_DIOW pulse. No further ATAPI_DIOR or ATAPI_DIOW pulses are asserted for this burst. If the device is able to continue the transfer of data, the device leaves the ATAPI_DMARQ asserted and waits for the host to re-assert ATAPI_DMACK or negates ATAPI_DMARQ at any time after detecting that ATAPI_DMACK is negated.

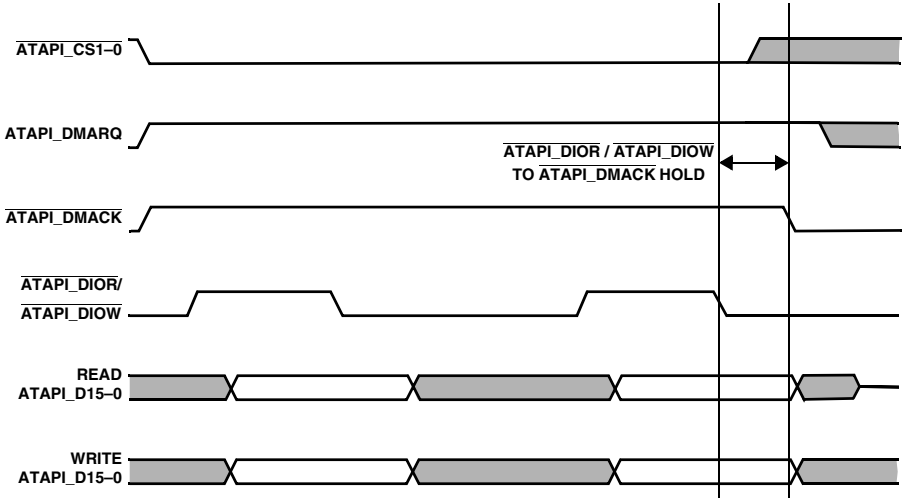


Figure 21-11. Host Terminating a Multiword DMA

Host Ultra DMA Block Implementation

The following steps occur during Ultra DMA-IN transfers.

Initiating an Ultra DMA Data-In Burst

1. The host keeps ATAPI_DMACK in the negated state before an Ultra DMA burst is initiated.
2. The device asserts ATAPI_DMARQ to initiate an Ultra DMA burst when ATAPI_DMACK is negated. After assertion of ATAPI_DMARQ the device does not negate ATAPI_DMARQ until after the first negation of ATAPI_DSTROBE.
3. Steps (c), (d), and (e) may occur in any order or at the same time. The host asserts ATAPI_STOP.
4. The host negates ATAPI_HDMARDY.

5. The host negates $\overline{\text{ATAPI\_CS1-0}}$ and ADDR3-1. The host keeps $\overline{\text{ATAPI\_CS1-0}}$ and ADDR3-1 negated until after negating $\overline{\text{ATAPI\_DMACK}}$ at the end of the burst.

6. Steps (c), (d), and (e) occurred at least t_{ACK} before the host asserts $\overline{\text{ATAPI\_DMACK}}$. The host keeps $\overline{\text{ATAPI\_DMACK}}$ asserted until the end of an Ultra DMA burst.

7. The host releases D15-0 within t_{AZ} after asserting $\overline{\text{ATAPI\_DMACK}}$.

8. The device may assert ATAPI_DSTROBE t_{ZIORDY} after the host has asserted $\overline{\text{ATAPI\_DMACK}}$. Once the device has driven ATAPI_DSTROBE the device does not release ATAPI_DSTROBE until after the host has negated $\overline{\text{ATAPI\_DMACK}}$ at the end of an Ultra DMA burst.

9. The host negates ATAPI_STOP and assert $\overline{\text{ATAPI\_HDMARDY}}$ within t_{ENV} after asserting $\overline{\text{ATAPI\_DMACK}}$. After negating ATAPI_STOP and asserting $\overline{\text{ATAPI\_HDMARDY}}$, the host does not change the state of either signal until after receiving the first negation of ATAPI_DSTROBE from the device (for example, after the first data word is received).

10. The device drives ATAPI_D15-0 no sooner than t_{ZAD} after the host has asserted $\overline{\text{ATAPI\_DMACK}}$, negated ATAPI_STOP, and asserted $\overline{\text{ATAPI\_HDMARDY}}$.

11. The device drives the first word of the data transfer onto D15-0. This step may occur when the device first drives D15-0 in step (j).

12. To transfer the first word of data the device negates ATAPI_DSTROBE within t_{FS} after the host has negated ATAPI_STOP and asserted $\overline{\text{ATAPI\_HDMARDY}}$. The device negates ATAPI_DSTROBE no sooner than t_{DVS} after driving the first word of data onto ATAPI_D15-0.

In Figure 21-12, the definitions for the ATAPI_DIOW, ATAPI_STOP, ATAPI_DIOR, ATAPI_HDMARDY, ATAPI_HSTROBE, ATAPI_IORDY, ATAPI_DDMARDY, and ATAPI_DSTROBE signal lines are not in effect until ATAPI_DMARQ and ATAPI_DMACK are asserted.

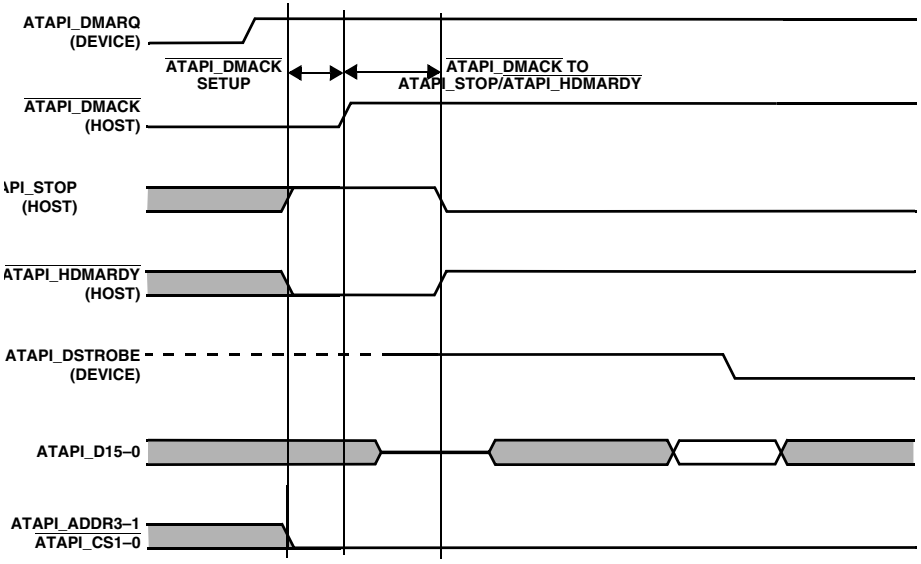


Figure 21-12. Initiating an Ultra DMA Data-In Burst

In Figure 21-13, the ATAPI_D15-0 and ATAPI_DSTROBE signals are shown at both the host and the device to emphasize that cable settling time, as well as cable propagation delay does not allow the data signals to be considered stable at the host until some time after they are driven by the device. See "Data-In Transfer" on page 21-30.

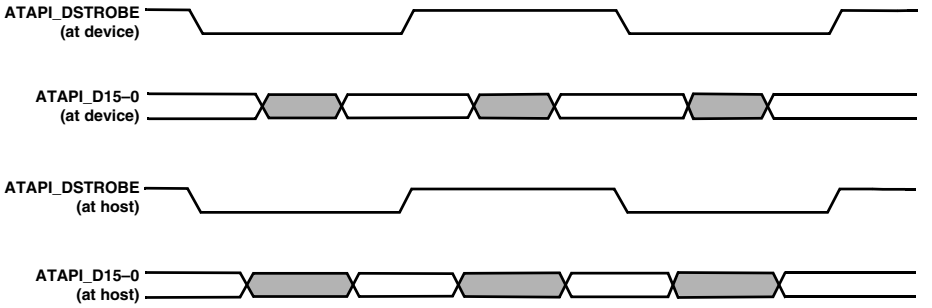


Figure 21-13. Sustaining a Ultra DMA Data IN Burst

Data-In Transfer

1. The device drives a data word onto ATAPI_D15-0.
2. The device generates a ATAPI_DSTROBE edge to latch the new word no sooner than t_{DVS} after changing the state of ATAPI_D15-0. The device generates a ATAPI_DSTROBE edge no more frequently than t_{CYC} for the selected Ultra DMA mode. The device does not generate two rising or two falling ATAPI_DSTROBE edges more frequently than t_{2CYC} for the selected Ultra DMA mode.
3. The device does not change the state of ATAPI_D15-0 until at least t_{DVH} after generating a ATAPI_DSTROBE edge to latch the data.
4. The device repeats steps (a), (b), and (c) until the Ultra DMA burst is paused or terminated by the device or host.

Device pausing an Ultra DMA Data-In Burst

1. The device does not pause an Ultra DMA burst until at least one data word of an Ultra DMA burst is transferred.
2. The device pauses an Ultra DMA burst by not generating additional ATAPI_DSTROBE edges
3. The device resumes an Ultra DMA burst by generating a ATAPI_DSTROBE edge.

Host pausing an Ultra DMA Data-In Burst

1. The host does not pause an Ultra DMA burst until at least one data word of an Ultra DMA burst is transferred.
2. The host pauses an Ultra DMA burst by negating $\overline{\text{ATAPI\_HDMARDY}}$.
3. The device stops generating ATAPI_DSTROBE edges within t_{RFS} of the host negating $\overline{\text{ATAPI\_HDMARDY}}$.
4. When operating in Ultra DMA modes 2, 1, or 0, the host is prepared to receive zero, one, or two additional data words after negating $\overline{\text{ATAPI\_HDMARDY}}$. While operating in Ultra DMA modes 5, 4, or 3, the host can receive zero, one, two, or three additional data words after negating $\overline{\text{ATAPI\_HDMARDY}}$. The additional data words are a result of cable round trip delay and t_{RFS} timing for the device.
5. The host resumes an Ultra DMA burst by asserting $\overline{\text{ATAPI\_HDMARDY}}$.

In Figure 21-14, the host may assert ATAPI_STOP to request termination of the ultra DMA burst no sooner than t_{RP} after $\overline{\text{ATAPI\_HDMARDY}}$ is negated. After negating $\overline{\text{ATAPI\_HDMARDY}}$, the host may receive zero, one, two, or three more data words from the device.

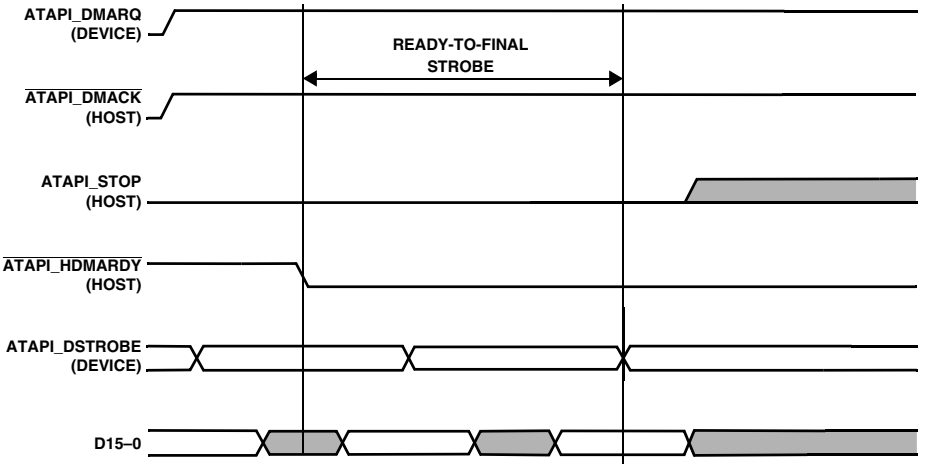


Figure 21-14. Host Pausing an Ultra DMA Data-In Burst

Ultra DMA Timing

Table 21-3. Ultra DMA Sender and Recipient Timing Parameters

Name	Description
$t_{2CYCTYP}$	Typical sustained average two-cycle time
t_{CYC}	Cycle time allowing for asymmetry and clock variations (from STROBE edge to STROBE edge)
t_{2CYC}	Two cycle time allowing for variations (from rising edge to next rising edge or from falling edge to next falling edge of STROBE)
t_{DS}	Data setup time at recipient (from data valid until STROBE edge)[1,2]
t_{DH}	Data hold time at recipient (from STROBE edge until data may become invalid)[1,2]
t_{DVS}	Data valid setup time at sender (from data valid until STROBE edge)[3]
t_{DVH}	Data valid hold time at sender (from STROBE edge until data may become invalid)[3]
t_{CS}	CRC word setup time at device[1]
t_{CH}	CRC word hold time device[1]
t_{CVS}	CRC word valid setup time at host (from CRC valid until $\overline{\texttt{ATAPI\_DMACK}}$ negation)[3]

Table 21-3. Ultra DMA Sender and Recipient Timing Parameters (Cont'd)

Name	Description
t_{CVH}	CRC word valid hold time at sender (from DMAC negation until CRC may become invalid)[3]
t_{ZFS}	Time from STROBE output released-to-driving until the first transition of critical timing
t_{DZFS}	Time from data output released-to-driving until the first transition of critical timing
t_{FS}	First STROBE time (for device to first negate ATAPI_DSTROBE from ATAPI_STOP during a data burst)
t_{LI}	Limited interlock time[4]
t_{MLI}	Interlock time with minimum[4]
t_{UI}	Unlimited interlock time[4]
t_{AZ}	Maximum time allowed for output drivers to release (from asserted or negated)
t_{ZAH}	Minimum delay time required for output
t_{ZAD}	Drivers to assert or negate (from released)
t_{ENV}	Envelope time (from $\overline{\texttt{ATAPI\_DMACK}}$ to ATAPI_STOP and $\overline{\texttt{ATAPI\_HDMARDY}}$ during data-in burst initiation and from $\overline{\texttt{ATAPI\_DMACK}}$ to ATAPI_STOP during data-out burst initiation)
t_{RFS}	Ready-to-final-STROBE time (no STROBE edges are sent this long after negation of $\overline{\texttt{ATAPI\_DDMARDY}}$)
t_{RP}	Ready-to-pause time (that recipient waits to pause after negating $\overline{\texttt{ATAPI\_DDMARDY}}$)
t_{IORDYZ}	Maximum time before releasing ATAPI_IORDY
t_{ZIORDY}	Minimum time before driving ATAPI_IORDY[5]
t_{ACK}	Setup and hold times for $\overline{\texttt{ATAPI\_DMACK}}$ (before assertion or negation)
t_{SS}	Time from STROBE edge to negation of ATAPI_DMARQ or assertion of ATAPI_STOP (when sender terminates a burst)
t_{DSIC}	Recipient IC data setup time (from data valid until STROBE edge)[6]
t_{DH}	Recipient IC data hold time (from STROBE edge until data becomes invalid)[6]

Table 21-3. Ultra DMA Sender and Recipient Timing Parameters (Cont'd)

Name	Description
t_{DVS}	Sender IC data valid setup time (from data valid until STROBE edge)[7]
t_{DVH}	Sender IC data valid hold time (from STROBE edge until data becomes invalid)[7]

1 80-conductor cabling (see Annex A) IS required in order to meet setup (t_{DS},t_{CS}) and hold (t_{DH},t_{CH}) time in modes greater than 2.

2 The parameters t_{DS} and t_{DH} for mode 5 are defined for a recipient at the end of the cable on line in a configuration with one device at the end of the cable.

3 Timing for t_{DVS}, t_{DVH}, t_{CVS}, and t_{CVH} shall be met for lumped capacitive loads of 15 and 40 pF at the connector where the Data and STROBE signals have the same capacitive load value. Due to reflections on the cable, these timing measurements are not valid in a normally functioning system.

4 The parameters t_{UI}, t_{MLI} and t_{LI} indicate sender-to-recipient or recipient-to-sender interlocks. For example, one agent (either sender or recipient) is waiting for the other agent to respond with a signal before proceeding. t_{UI} is an unlimited interlock that has no maximum time value. t_{MLI} is a limited time-out that has a defined minimum. t_{LI} is a limited time-out that has a defined maximum.

5 For all modes the parameter t_{ZIORDY} may be greater than t_{ENV} due to the fact that the host has a pull-up on ATAPI_IORDY giving it a known state when released.

6 The correct data value is captured by the recipient given input data with a slew rate of 0.4 V/ns (rising and falling) and the input STROBE with a slew rate of 0.4 V/ns (rising and falling) at t_{DSIC} and t_{DHIC} timing (as measured through 1.5 V).

7 The parameters t_{DVSIC} and t_{DVHIC} are met for lumped capacitive loads of 15 and 40 pF at the IC where all signals have the same capacitive load value. Noise that may couple onto the output signals by external sources in a normally functioning system has not been included in these values.

In Figure 21-15, the definitions for the ATAPI_STOP, $\overline{\text{ATAPI\_HDMARDY}}$, and ATAPI_DSTROBE signal lines are not in effect after ATAPI_DMARQ and $\overline{\text{ATAPI\_DMACK}}$ are negated. See"Device Terminating the Ultra DMA Data-In Transfer" on page 21-17

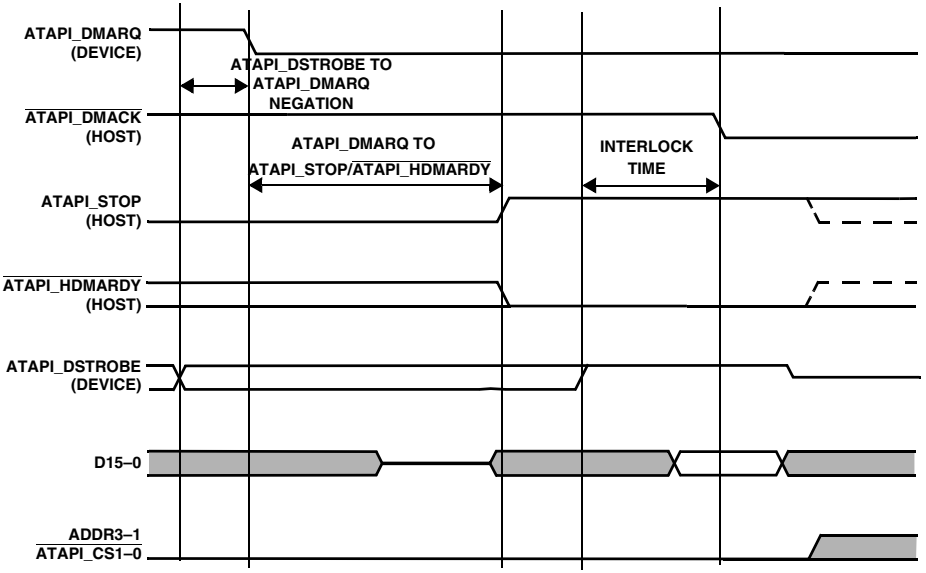


Figure 21-15. Device Terminating Ultra DMA Data-In Burst

In Figure 21-16, the definitions for the ATAPI_STOP, ATAPI_HDMARDY, and ATAPI_DSTROBE signal lines are not in effect after ATAPI_DMARQ and ATAPI_DMACK are negated. See "Host Terminating the Ultra DMA Data-In Transfer" on page 21-16.

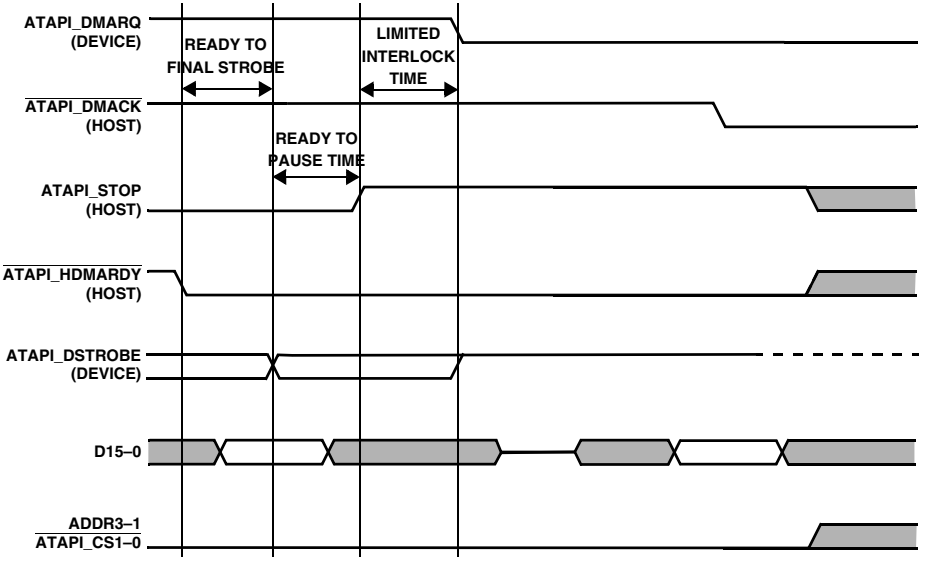


Figure 21-16. Host Terminating Ultra DMA Data-In Burst

Ultra DMA-Out Timing

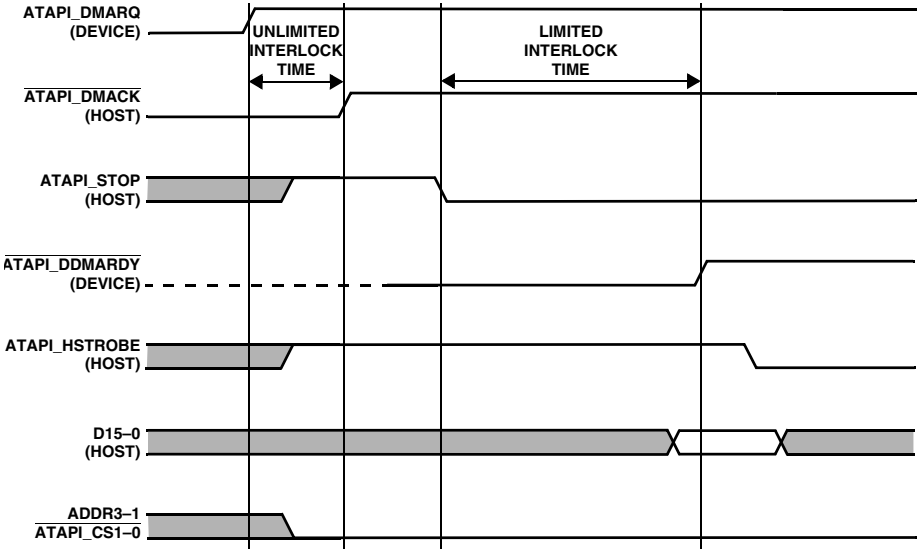


Figure 21-17. Initiating Ultra DMA Data-Out Burst

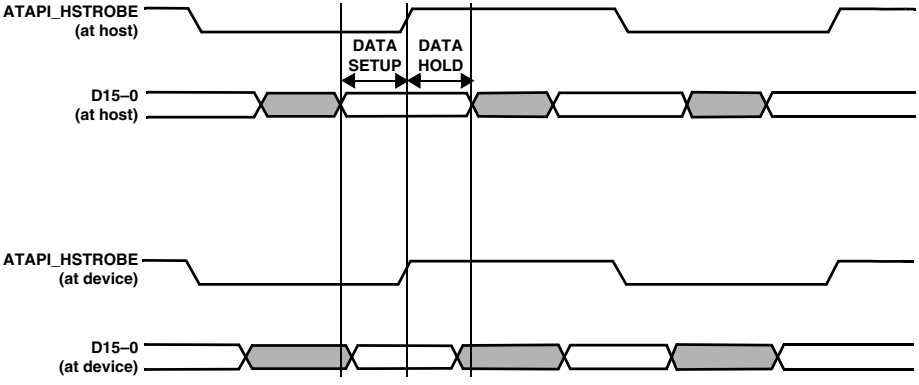


Figure 21-18. Sustaining Ultra DMA Data-Out Burst

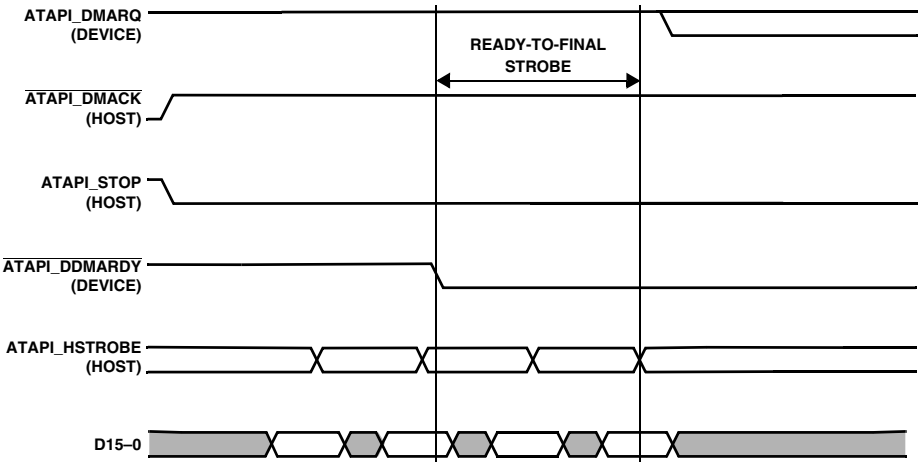


Figure 21-19. Device Pausing Ultra DMA Data-Out Burst

In Figure 21-20, the definitions for the ATAPI_STOP, ATAPI_DDMARDY, and ATAPI_HSTROBE signal lines are no longer in effect after ATAPI_DMARQ and ATAPI_DMACK are negated. See "Host Terminating Ultra DMA Data-Out Transfer" on page 21-17.

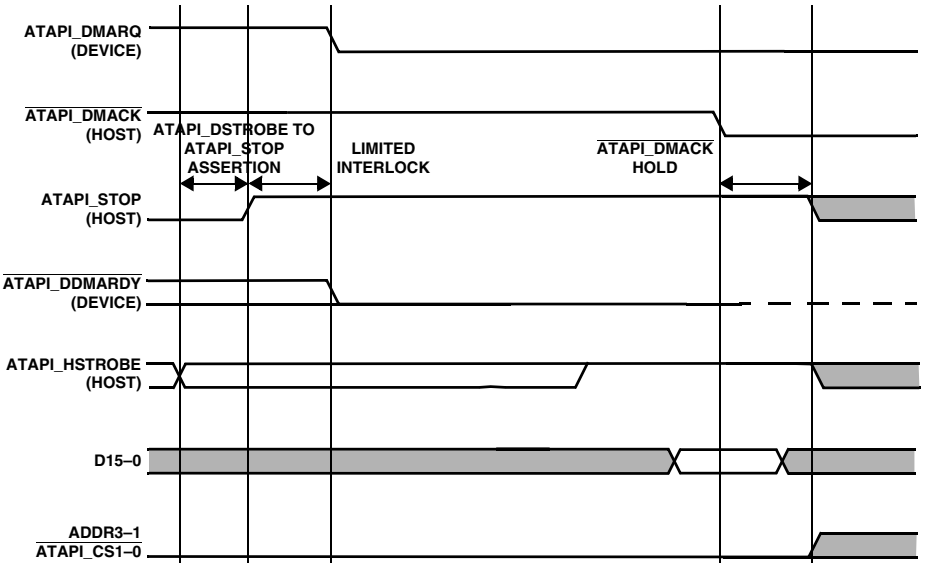


Figure 21-20. Host Terminating Ultra DMA Data-Out Burst

In Figure 21-21, the definitions for the ATAPI_STOP, ATAPI_DDMARDY, and ATAPI_HSTROBE signal lines are no longer in effect after ATAPI_DMARQ and ATAPI_DMACK are negated. See "Device Terminating the Ultra DMA Data-Out Transfer" on page 21-18.

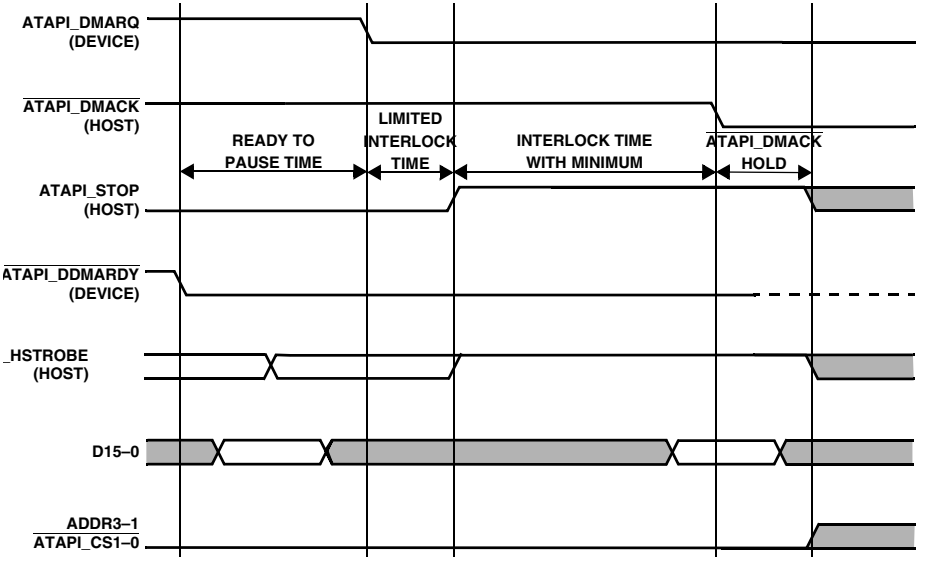


Figure 21-21. Device Terminating Ultra DMA Data OUT Burst

Programming Model

The following sections describe the ATAPI peripheral's programming model.

ATAPI Device Configuration and Setup

1. Detection of devices on ATAPI Cable
    - Power-On–Reset protocol
    - Execute device diagnostic command & hardware initialization
    - Read the device signature

2. Identifying the features of devices on ATAPI Cable
    - Select each device & execute IDENTIFY DEVICE Command
    - Read the signature of each device and decode the features supported
3. Selecting a Device, configuring the device and executing the commands
    - Select a device
    - Configure the device with the mode supported (PIO, DMA, ultra DMA)
    - Prepare the device, deliver & execute the command.

The basic data flow operation from ATAPI host to ATAPI device is described as follows:

- ATAPI host reads a pre-defined buffer descriptor, decodes it, and gets the length and start address.
- Fetches the data and processes it.
- Selects a device by doing a register write transfer for setting the `DEV` bit in the device control register.
- Writes all device command block parameters for the command (such as, sector count, LBA, features, among others)
- Splits the complete data for the current command in terms of DRQ blocks

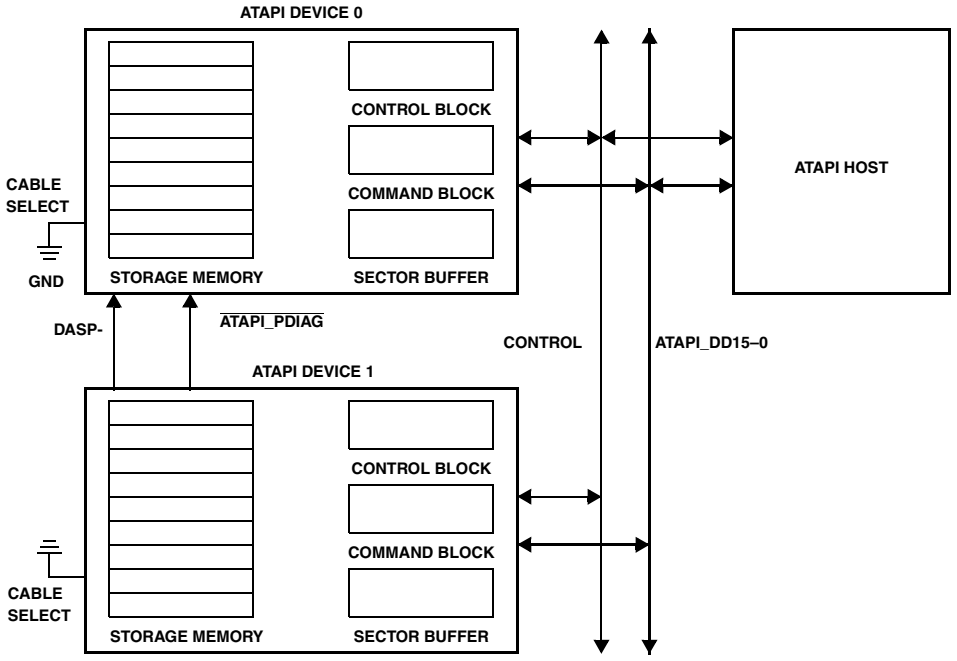


Figure 21-22. ATAPI Device and Host Configuration

- Polls the device status register (BSY bit) to check if device is ready for transfer (interrupt of device is disabled) or else waits for interrupt (ATAPI_INTRQ) from device and then reads the status register.
- Triggers the ATAPI controller for DMA transfers equal to the length of the DRQ block size to/from device data port (using PIO/DMA/Ultra DMA transfers) or vice-versa. This is repeated until all the data for the current command is completed.

PIO Data-out Transfers Pseudo-code

```
//Select the Device
Read Device register (Device Control, Dev bit);
If (not selected)
  Write device register (Device Control, Dev bit);
  Read Device register (Device Control, Dev bit);

// Initialize the Device parameters for the command
Write Device Parameters
     (Sector count Register: Command Data size;
      Feature Register: Specific Data;
      LBA High Register: yy;
      LBA Mid Register: xx;
      LBA Low Register: zz;
      )

// Write Device Command
Write Device Register (Command Register, WRITE SECTOR);

// Start Data Transfer
     // Check if the device is ready for data transfer
     Check Device Register (Status Register, bsy=0, drq = 1);
     Start DMA (length: DRQ blockn, addr: x0,
                pio_start: 1, xfer_dir: 1)

 Check Device Register (Status Register, bsy =0, drq =0);

// Command Completed
```

Host Multiword DMA Transfers Pseudo-code

```
// Select the Device
  Read Device register (Device Status);
  If (not selected)
  Write device register (Device Control, Dev bit);
  Read Device register (Device Status);
// Initialize the Device parameters for the command
  Write Device Parameters (Sector count Register: Command Data
  size;
  Feature Register: Specific Data;
  LBA High Register: yy;
  LBA Mid Register: xx;
  LBA Low Register: zz;
  )
// Write Device Command
  Write Device Register (Command Register, WRITE DMA);
// Start Data Transfer
  While (Data Transferred < command data size)
  {
    // Check the device is ready for data transfer
    Check Device Register (Status Register, (bsy=0, drq = 1) or
    (bsy=1,drq=0));
    Start DMA (length: DRQ block1, addr: xxxx,
    ultra_start: 1, xfer_dir: 1)
  }
  Wait for INTRQ_wait () // Host input flag checking
  Check Device Register (Status Register, (bsy=0, drq = 0) ;
```

Host Ultra DMA Command Protocol Transfers Pseudo-code

```
Prepare_device (cmd, length, tfr_type);

 {
        device_register_write ( sector_count_reg, length);
        device_register_write ( lbah_reg, lbah);
        device_register_write ( lbam_reg, lbam);
        device_register_write ( lbal_reg, lbal);
        device_register_write (command_reg, cmd);
 }

cur_len = length;

while(cur_len > cur_dmasize)
    {
         {
            do_tx ( cur_dmasize , cur_dmem_addr, tfr_type,
last_burst=0);
            write_pio (DEV_ADDR, 0);
            write_pio (DMEM_LEN, cur_dmasize);
            write_pio (DMEM_ADDR, cur_dmem_addr);
            write_pio (ATAPI_CONTROL,
                       xfer_dir_bit,ULTRA_OUT_START);
            wait for ATAPI_DONE_FLAG;
        }
         cur_len = cur_len - cur_dmasize);
         cur_dmem_addr = cur_dmem_addr + cur_dmasize;
  }

 if (cur_len != 0)
   do_tx ( cur_dmasize , cur_dmem_addr, tfr_type, last_burst=1);
```

ATAPI Registers

The ATAPI interface's memory-mapped registers (MMRs) regulate its operation. Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table 21-4 lists the ATAPI memory-mapped registers, starting at base address 0xFFC03800. Register addresses are given relative to the base address.

Table 21-4. ATAPI Core Registers

Address	Register Name	Description
	ATAPI Control Registers	
0xFFC0 3800	ATAPI_CONTROL	"ATAPI Control (ATAPI_CONTROL) Register" on page 21-49
0xFFC0 3804	ATAPI_STATUS	"ATAPI Status (ATAPI_STATUS) Register" on page 21-51
0xFFC0 3808	ATAPI_DEV_ADDR	"ATAPI Device Address (ATAPI_DEV_ADDR) Register" on page 21-52
0xFFC0 380C	ATAPI_DEV_TXBUF	"ATAPI Device Transmit Buffer (ATAPI_DEV_TXBUF) Register" on page 21-53
0xFFC0 3810	ATAPI_DEV_RXBUF	"ATAPI Device Receive Buffer (ATAPI_DEV_RXBUF) Register" on page 21-54
0xFFC0 3814	ATAPI_INT_MASK	"ATAPI Interrupt Mask (ATAPI_INT_MASK) Register" on page 21-54
0xFFC0 3818	ATAPI_INT_STATUS	"ATAPI Interrupt Status (ATAPI_INT_STATUS) Register" on page 21-56
0xFFC0 381C	ATAPI_XFER_LEN	"ATAPI Transfer Length (ATAPI_XFER_LEN) Register" on page 21-58

Table 21-4. ATAPI Core Registers (Cont'd)

Address	Register Name	Description
0xFFC0 3820	ATAPI_LINE_STATUS	"ATAPI Line Status (ATAPI_LINE_STATUS) Register" on page 21-59
0xFFC0 3824	ATAPI_SM_STATE	"ATAPI State Machine Status (ATAPI_SM_STATE) Register" on page 21-59
0xFFC0 3828	ATAPI_TERMINATE	"ATAPI Host Terminate (ATAPI_TERMINATE) Register" on page 21-60
0xFFC0 382C	ATAPI_PIO_TFRCNT	"ATAPI PIO Transfer Count (ATAPI_PIO_TFRCNT) Register" on page 21-61
0xFFC0 3830	ATAPI_DMA_TFRCNT	"ATAPI Multiword DMA Transfer Count (ATAPI_MULTI_TFRCNT) Register" on page 21-61
0xFFC0 3834	ATAPI_ULTRA_IN_TFRCNT	"ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT) Register" on page 21-62
0xFFC0 3838	ATAPI_ULTRA_OUT_TFRCNT	"ATAPI Ultra DMA OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT) Register" on page 21-63
	PIO and REG Mode Registers	
0xFFC0 3840	ATAPI_REG_TIM_0	"ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0) Register" on page 21-63
0xFFC0 3844	ATAPI_PIO_TIM_0	"ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0) Register" on page 21-64
0xFFC0 3848	ATAPI_PIO_TIM_1	"ATAPI Programmed I/O Timing 1 (ATAPI_PIO_TIM_1) Register" on page 21-64

Table 21-4. ATAPI Core Registers (Cont'd)

Address	Register Name	Description
	Multi-DMA Mode Registers	
0xFFC0 3850	ATAPI_MULTI_TIM_0	"ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0) Register" on page 21-65
0xFFC0 3854	ATAPI_MULTI_TIM_1	"ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1) Register" on page 21-65
0xFFC0 3858	ATAPI_MULTI_TIM_2	"ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2) Register" on page 21-66
	Ultra-DMA Mode Registers	
0xFFC0 3860	ATAPI_ULTRA_TIM_0	"ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0) Register" on page 21-66
0xFFC0 3864	ATAPI_ULTRA_TIM_1	"ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1) Register" on page 21-67
0xFFC0 3868	ATAPI_ULTRA_TIM_2	"ATAPI Ultra DMA Timing 2 (ATAPI_ULTRA_TIM_2) Register" on page 21-67
0xFFC0 386C	ATAPI_ULTRA_TIM_3	"ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3) Register" on page 21-68

ATAPI Control and Status Registers

This section describes the details of the ATAPI core registers.

ATAPI Control (ATAPI_CONTROL) Register

The ATAPI_CONTROL register (see Figure 21-23) starts, stops, and selects termination handling for ATAPI data transfers.

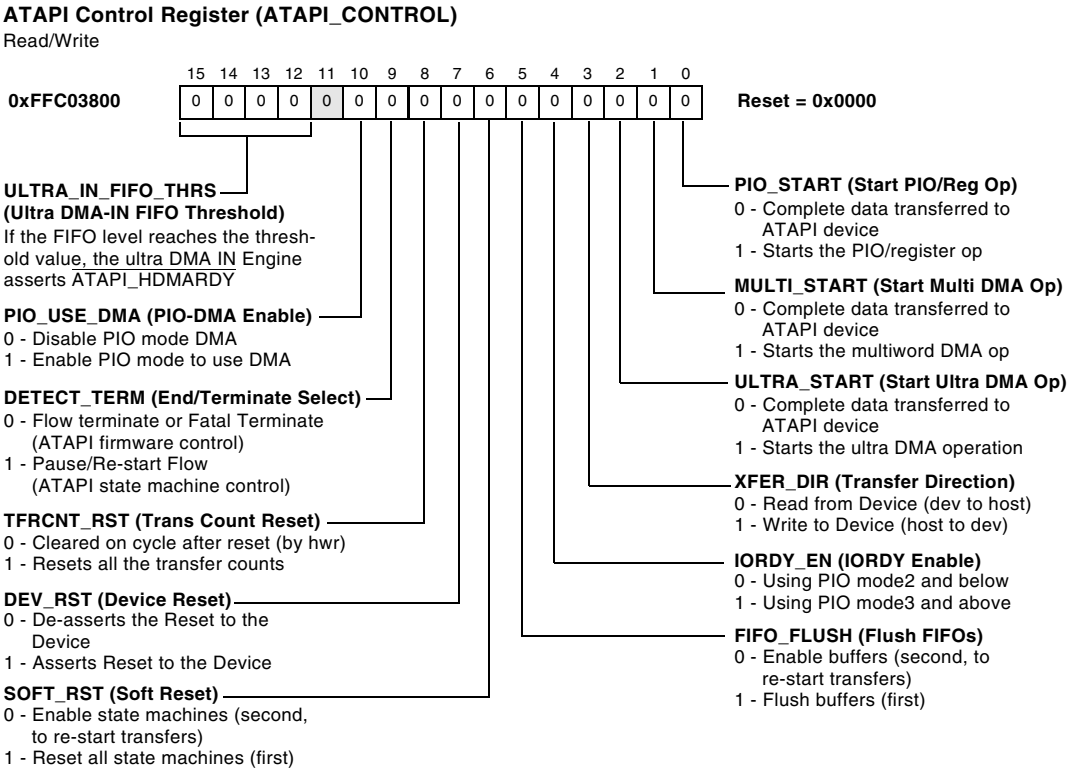


Figure 21-23. ATAPI Control Register

The PIO_START, MULTI_START, and ULTRA_START bits start transfer operations. These bits are reset by the ATAPI host only after the complete data is transferred on the ATAPI device or when an error occurs during the transfer.

The FIFO_FLUSH bit flushes the various FIFOs in the system to a known state. This flush may be required if some data remains in the FIFO because of early termination of transfers. The bit should be set to flush the FIFO and then cleared by the firmware to restart the transfers.

The ATAPI_CONTROL register includes a number of reset bits. The SOFT_RST bit resets all state machines of the ATAPI host independent of the ATAPI device state. Firmware sets the SOFT_RST bit to reset all state machines, then firmware clears the SOFT_RST bit to restart the transfers. The DEV_RST bit, when set, asserts a reset to the ATAPI device. The DEV_RST bit must be cleared to deassert the reset. The TFRCNT_RST bit resets all the transfer counts. The host firmware asserts TFRCNT_RST to reset all the transfer counts, and the hardware clears the TFRCNT_RST bit in the next cycle.

The END_ON_TERM bit selects operation when a device terminate sequence occurs and selects whether the ATAPI host or firmware controls the restart of the transfer. When END_ON_TERM is set (=1), if the device initiates the terminate sequence before the complete data for the command is transferred, the ATAPI host state machine waits in its intermediate state for the device response to restart the transfer for the remaining data to be transferred. If END_ON_TERM is cleared (=0), if the device initiates the terminate sequence before the complete data for the command is transferred, the ATAPI host state machine goes to the idle state and asserts the MULTI_TERM_INT flag in the ATAPI interrupt status register along and updates the corresponding transfer count. This gives control to the ATAPI firmware to decide further operation. The ATAPI firmware can then read the device status register to know whether it was a flow terminate or a fatal terminate.

The PIO_USE_DMA bit is set to enable PIO mode to use DMA. By default, PIO DMA usage is disabled and data transfer in PIO mode happens by writing into the ATAPI_DEV_TXBUF register and performing one transfer at a time.

The ULTRA_IN_FIFO_THRS bits select the ultra DMA input FIFO threshold. If the FIFO level reaches the threshold value, the ultra DMA input engine asserts the $\overline{\text{ATAPI\_HDMARDY}}$ pin to signal to the device to stop transferring the data.

ATAPI Status (ATAPI_STATUS) Register

The ATAPI_STATUS register (see Figure 21-24) provides status information on ATAPI data transfers in progress.

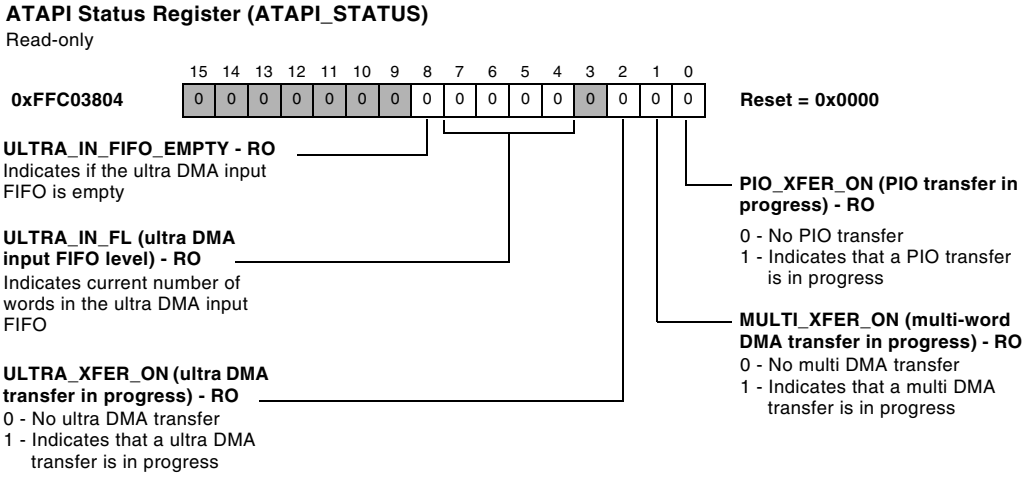


Figure 21-24. ATAPI Status Register

ATAPI Device Address (ATAPI_DEV_ADDR) Register

The ATAPI_DEV_ADDR register (see Figure 21-25) selects the ATAPI device address.

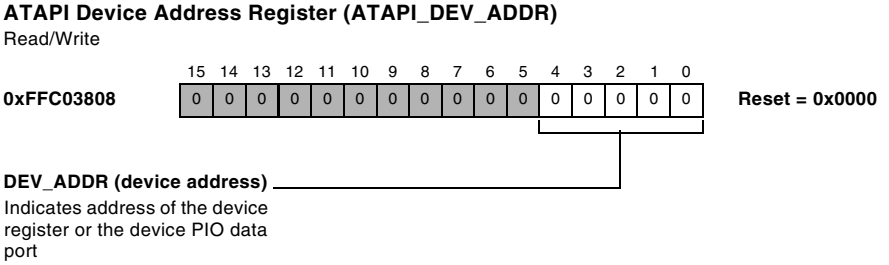


Figure 21-25. ATAPI Device Address Register

The DEV_ADDR bits contain the address of the device register or the device PIO data port. Based on this address, the ATAPI block decides whether to perform a PIO data port operation or device register operation.

Table 21-5. DEV_ADDR Bit Field Value Ranges

Address Value	Description
0x00	PIO / DMA /Ultra DMA Data port
0x01 – 0x07	Device Command Block Registers
0x08 – 0x0F	Device Control Block Registers

The ATAPI host firmware should program the ATAPI_DEV_ADDR register with the address of the device register, which is being accessed.

Table 21-6. ATAPI_DEV_ADDR Register Address Values

Address Value	Description
0x01	Error/Feature
0x02	Sector Count
0x03	LBA (low)
0x04	LBA (mid)
0x05	LBA (high)
0x06	Device
0x07	Status/Command
0x0E	Alternate Status/Device Control

ATAPI Device Transmit Buffer (ATAPI_DEV_TXBUF) Register

The `ATAPI_DEV_TXBUF` register (see Figure 21-26) holds write data for the ATAPI device register write transfers.

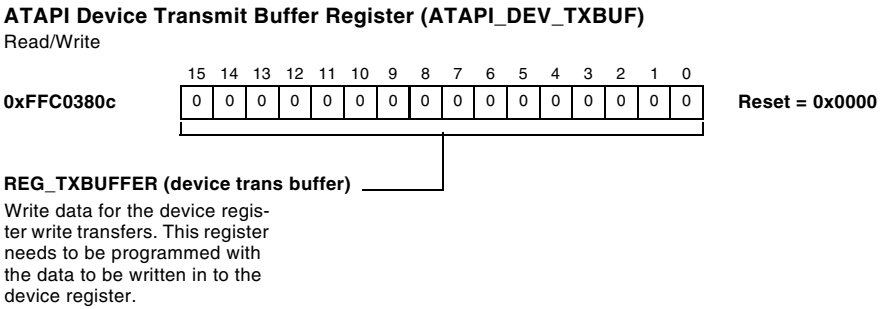


Figure 21-26. ATAPI Device Transmit Buffer Register

ATAPI Device Receive Buffer (ATAPI_DEV_RXBUF) Register

The ATAPI_DEV_RXBUF register (see Figure 21-26) holds receive data for the ATAPI device register read transfers.

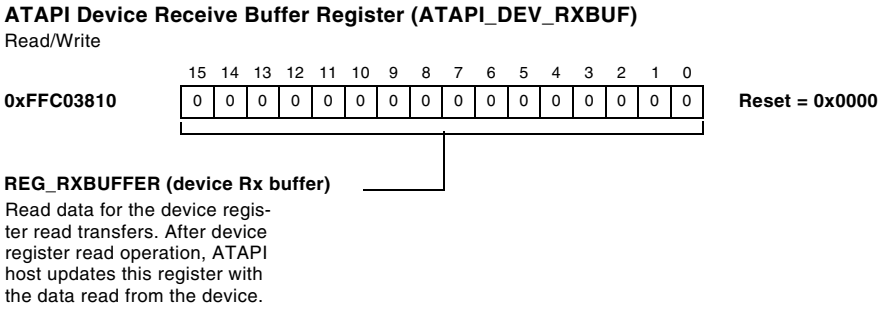


Figure 21-27. ATAPI Device Receive Buffer Register

ATAPI Interrupt Mask (ATAPI_INT_MASK) Register

The ATAPI_INT_MASK register (see Figure 21-28) enables interrupt sources to assert the interrupt output. Each mask bit corresponds to one interrupt source bit in the ATAPI interrupt status (ATAPI_INT_STAT) register. For more information about these interrupts, see "ATAPI Interrupt Status (ATAPI_INT_STATUS) Register" on page 21-56.

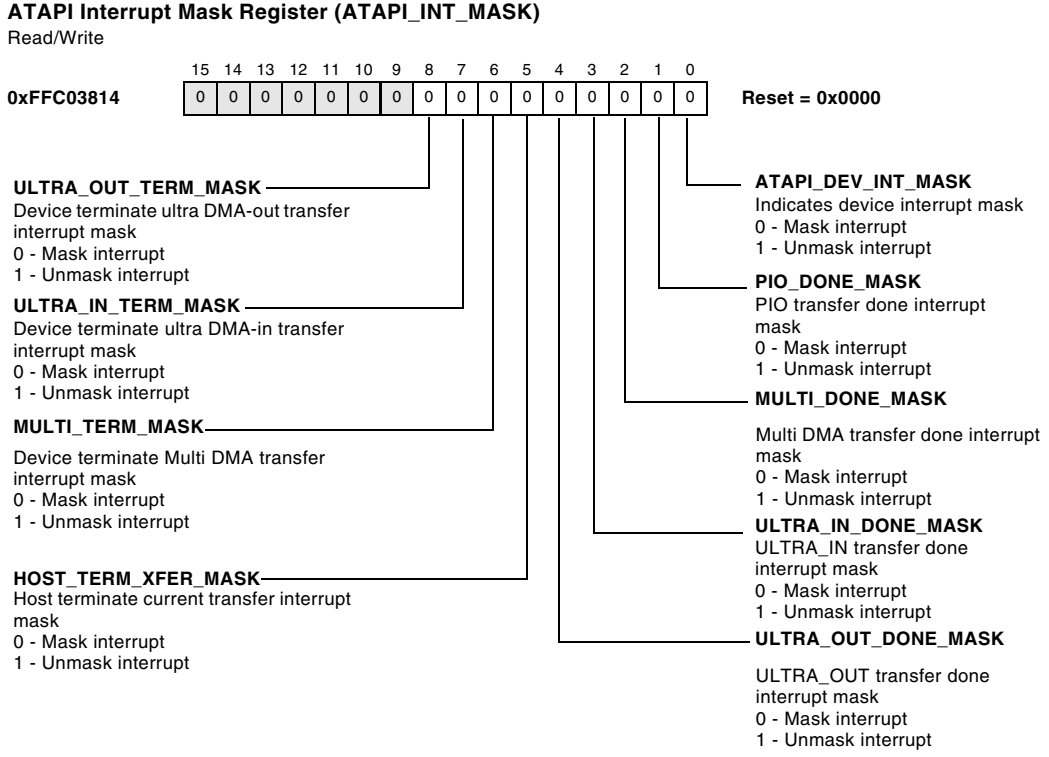


Figure 21-28. ATAPI Interrupt Mask Register

ATAPI Interrupt Status (ATAPI_INT_STATUS) Register

The ATAPI_INT_STATUS register (see Figure 21-29) contains information about functional areas that require service.

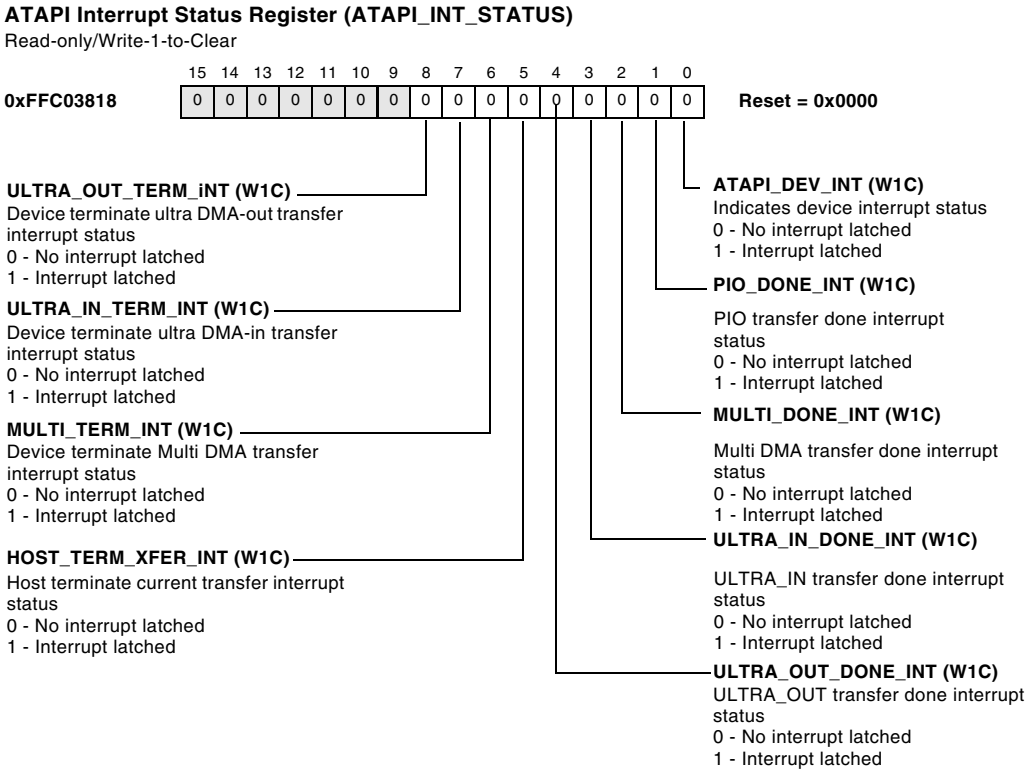


Figure 21-29. ATAPI Interrupt Status Register

After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit. ATA_DEV_INT is the interrupt generated by the device. The rest of the interrupts are generated by the host. Either the device or the host interrupt can be used by the firmware. If the corresponding interrupt mask bit in the ATAPI_INT_STAT register is not set,

there is no interrupt generated. For more information about masking these interrupts, see "ATAPI Interrupt Mask (ATAPI_INT_MASK) Register" on page 21-54.

The ATAPI_DEV_INT (W1C) bit indicates an ATAPI device interrupt is asserted on the ATAPI interface by the device. It is cleared by writing a 1.

The PIO_DONE_INT, MULTI_DONE_INT, ULTRA_IN_DONE_INT, and ULTRA_OUT_DONE_INT (W1C) bits indicate that interrupts have been asserted on completion of various types of transfers.

The HOST_TERM_XFER_INT (W1C) bit indicates that the interrupt is asserted on host termination of the current transfer.

The MULTI_TERM_INT (W1C) bit indicates that the interrupt is asserted on device termination of the multiword DMA transfer. The MULTI_TERM_INT bit is set when the device initiates a termination sequence before the complete data is transferred in multiword DMA mode (for example, when programmed XFER_LEN of ATA words have not been transferred across the device). If DETECT_TERM is not set, the control is passed on to the firmware, and the firmware can read the device status register to detect the reason for early termination.

The ULTRA_IN_TERM_INT and ULTRA_OUT_TERM_INT (W1C) bits indicate that interrupts have been asserted on device termination of ultra DMA in or out transfers. The ULTRA_IN_TERM_INT or ULTRA_OUT_TERM_INT bits are set when the device initiates a termination sequence before the complete data is transferred in ultra DMA in or out mode (for example, when programmed XFER_LEN of ATA words have not been transferred across the device). If DETECT_TERM is not set, control is passed on to the firmware, and the firmware can read the device status register to detect the reason for early termination.

ATAPI Transfer Length (ATAPI_XFER_LEN) Register

The ATAPI_XFER_LEN register (see Figure 21-30) holds the transfer length in number of ATA words (1 ATA word = 2 bytes). This register needs to be programmed with the number of ATA words that need to be transferred from device to host or vice versa. This register value is used for all three transfer modes – PIO, DMA, and ultra DMA. As the transfer progresses, this register is constantly updated with the number of ATA words that are pending to be transferred.

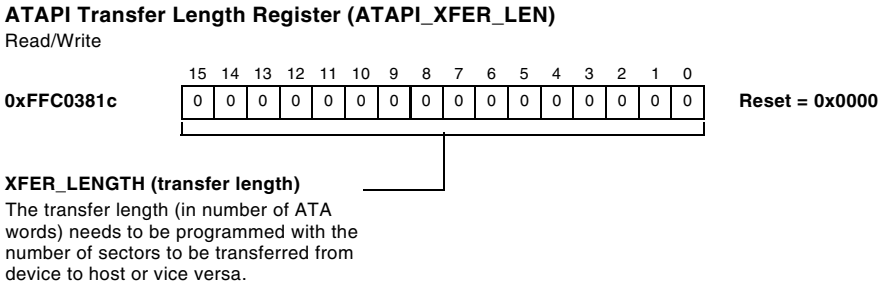


Figure 21-30. ATAPI Transfer Length Register

ATAPI Line Status (ATAPI_LINE_STATUS) Register

The ATAPI_LINE_STATUS register (see Figure 21-31) provides line status information on the ATAPI interface activity.

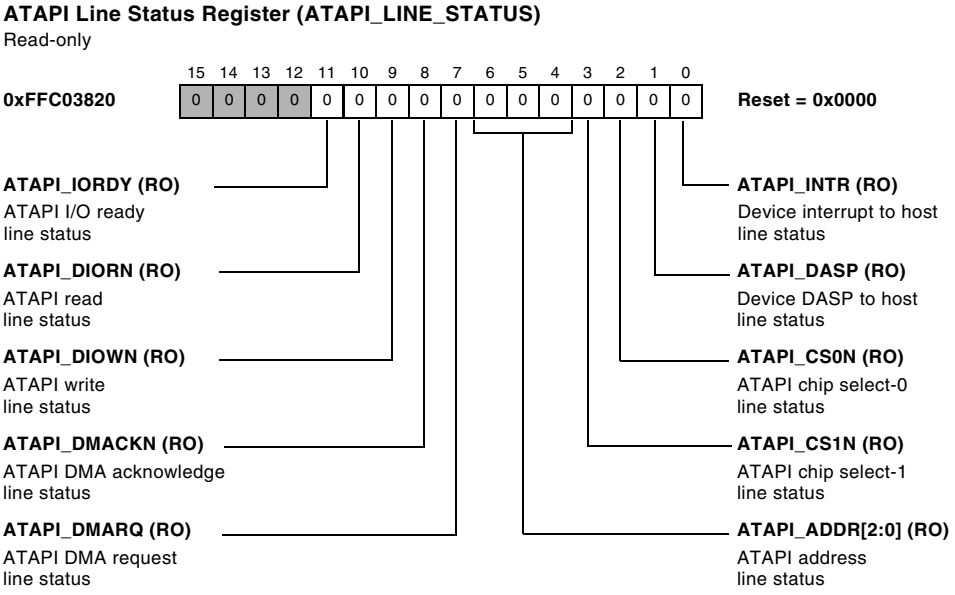


Figure 21-31. ATAPI Line Status Register

ATAPI State Machine Status (ATAPI_SM_STATE) Register

The ATAPI_SM_STATE register (see Figure 21-32) provides state machine status information on the ATAPI interface.

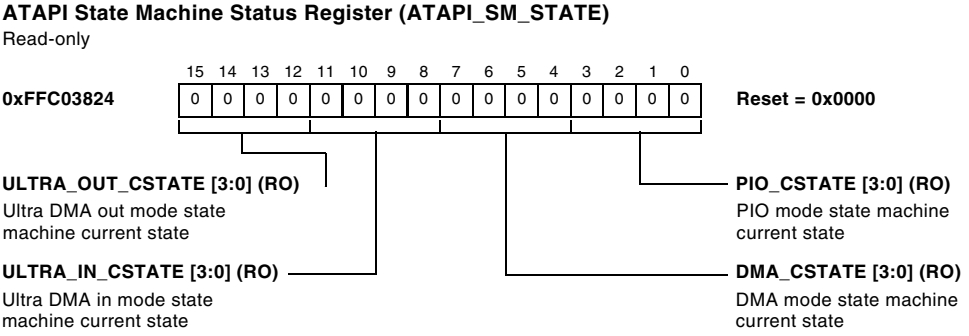


Figure 21-32. ATAPI State Machine Status Register

ATAPI Host Terminate (ATAPI_TERMINATE) Register

When set to 1, the `ATAPI_TERMINATE` register (see Figure 21-33) initiates a terminate sequence on the device. Once the termination sequence is over, bit 0 is reset by the hardware. The ATAPI host firmware should wait until this bit is cleared before taking any further operation, as the termination sequence takes some time depending upon the device response.

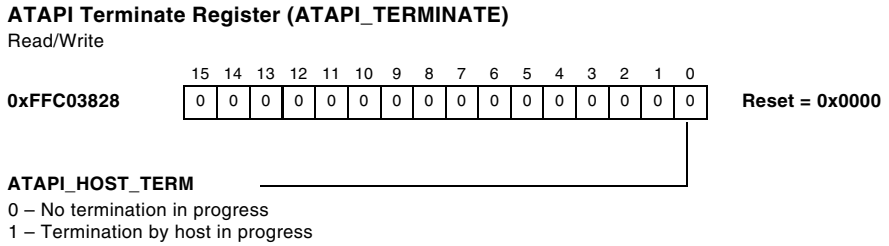


Figure 21-33. ATAPI Terminate Register

ATAPI PIO Transfer Count (ATAPI_PIO_TFRCNT) Register

The ATAPI_PIO_TFRCNT register (see Figure 21-34) indicates the PIO transfer count. This count indicates the transfer count of ATA words transferred across the device for the current DMA burst in PIO mode. The count gets cleared by setting the TFRCNT_RST bit in the ATAPI_CONTROL register. If the TFRCNT_RST bit was not set with the start of DMA burst, the transfer count continues from the previous value.

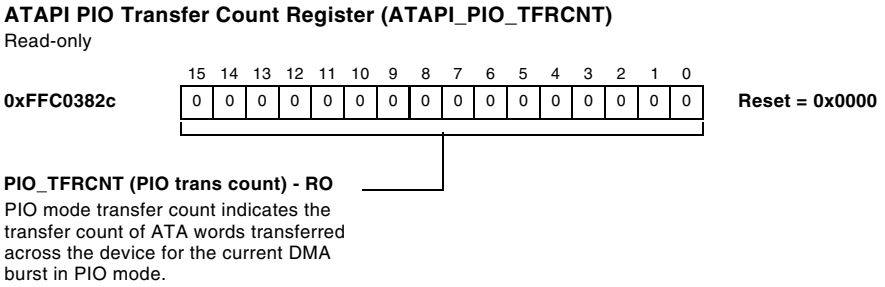


Figure 21-34. ATAPI PIO Transfer Count Register

ATAPI Multiword DMA Transfer Count (ATAPI_MULTI_TFRCNT) Register

The ATAPI_MULTI_TFRCNT register (see Figure 21-35) indicates the transfer count of ATA words transferred across the device for the current DMA burst in multiword DMA mode. The count gets cleared by setting the TFRCNT_RST bit in the ATAPI_CONTROL register. If the TFRCNT_RST bit is not set with the start of the DMA burst, the transfer count continues from the previous value.

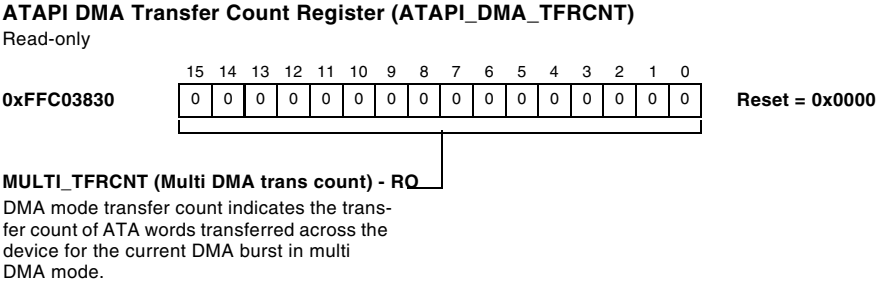


Figure 21-35. ATAPI DMA Transfer Count Register

ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT) Register

The ATAPI_ULTRA_IN_TFRCNT register (see Figure 21-36) indicates ultra DMA in mode transfer count of ATA words transferred across the device for the current DMA burst in ultra DMA in mode. The count gets cleared by setting the TFRCNT_RST bit in the ATAPI_CONTROL register. If the TFRCNT_RST bit is not set with the start of DMA burst, the transfer count continues from the previous value.

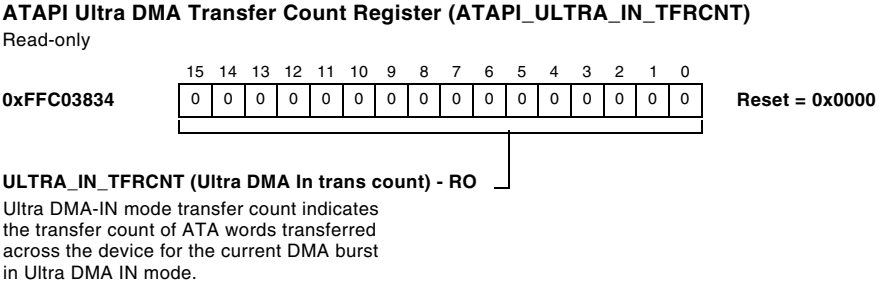


Figure 21-36. ATAPI Ultra DMA Transfer Count Register

ATAPI Ultra DMA OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT) Register

The ATAPI_ULTRA_OUT_TFRCNT register (see Figure 21-37) indicates ultra DMA in mode transfer count of ATA words transferred across the device for the current DMA burst in ultra DMA in mode. The count gets cleared by setting the TFRCNT_RST bit in the ATAPI_CONTROL register. If the TFRCNT_RST bit is not set with the start of DMA burst, the transfer count continues from the previous value.

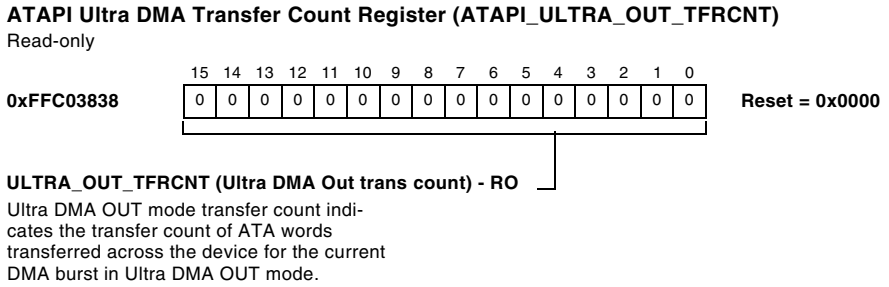


Figure 21-37. ATAPI ULTRA_OUT Transfer Count Register

ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0) Register

The ATAPI_REG_TIM_0 register (see Figure 21-38) holds timing parameter settings (in terms of system clock counts) for register transfer operations.

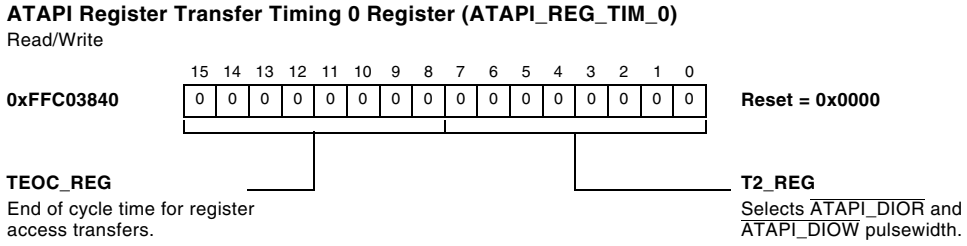


Figure 21-38. ATAPI Register Transfer Timing 0 Register

ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0) Register

The ATAPI_PIO_TIM_0 register (see Figure 21-39) holds timing parameter settings (in terms of system clock counts) for programmed I/O operations.

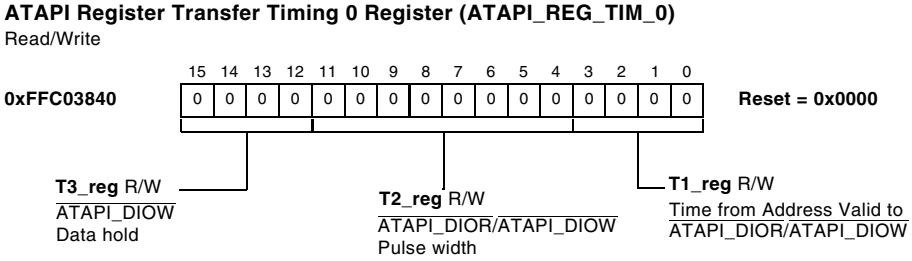


Figure 21-39. ATAPI Programmed I/O Timing 0 Register

ATAPI Programmed I/O Timing 1 (ATAPI_PIO_TIM_1) Register

The ATAPI_PIO_TIM_1 register (see Figure 21-40) holds timing parameter settings (in terms of system clock counts) for programmed I/O operations. The value of TEOC is T0-T2.

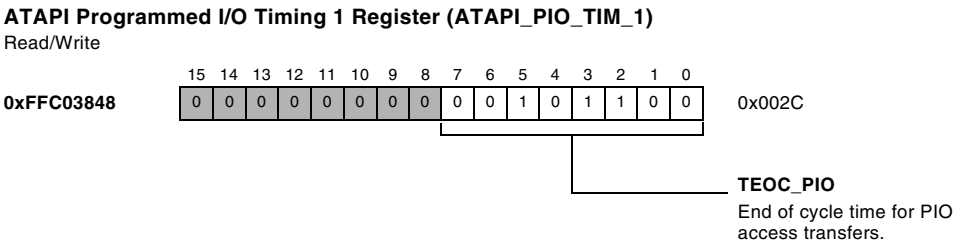


Figure 21-40. ATAPI Programmed I/O Timing 0 Register

ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0) Register

The ATAPI_MULTI_TIM_0 register (see Figure 21-41) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations.

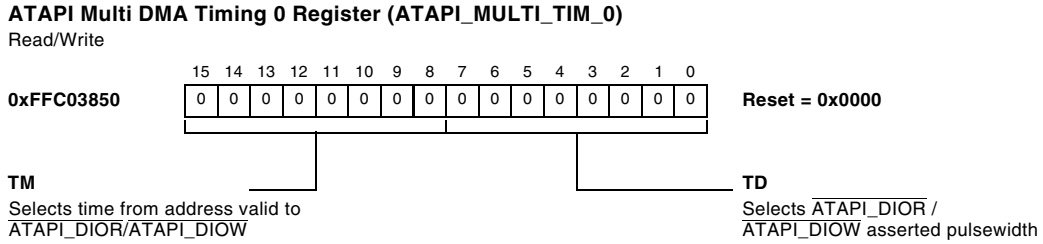


Figure 21-41. ATAPI Multi DMA Timing 0 Register

ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1) Register

The ATAPI_MULTI_TIM_1 register (see Figure 21-42) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations.

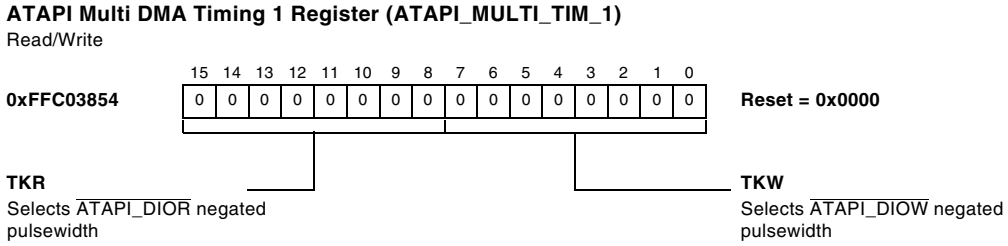


Figure 21-42. ATAPI Multi DMA Timing 1 Register

ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2) Register

The ATAPI_MULTI_TIM_2 register (see Figure 21-43) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations. The value of TEOC is Tj.

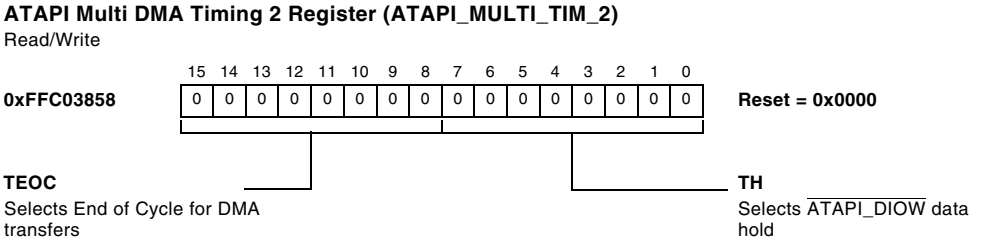


Figure 21-43. ATAPI Multi DMA Timing 2 Register

ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0) Register

The ATAPI_ULTRA_TIM_0 register (see Figure 21-44) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

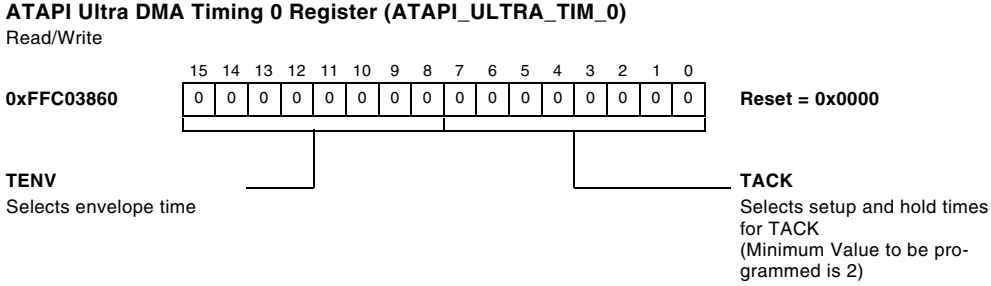


Figure 21-44. ATAPI Ultra DMA Timing 0 Register

ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1) Register

The ATAPI_ULTRA_TIM_1 register (see Figure 21-45) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

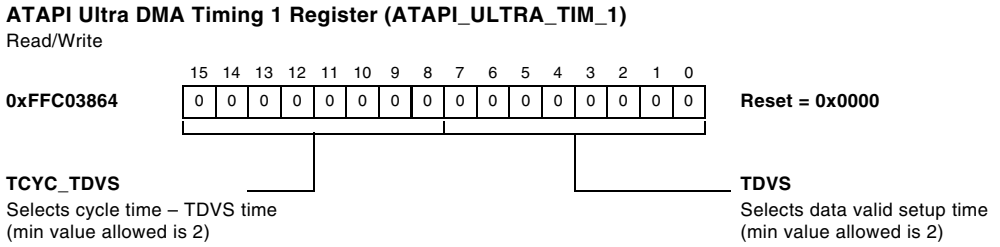


Figure 21-45. ATAPI Ultra DMA Timing 1 Register

ATAPI Ultra DMA Timing 2 (ATAPI_ULTRA_TIM_2) Register

The ATAPI_ULTRA_TIM_2 register (see Figure 21-46) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

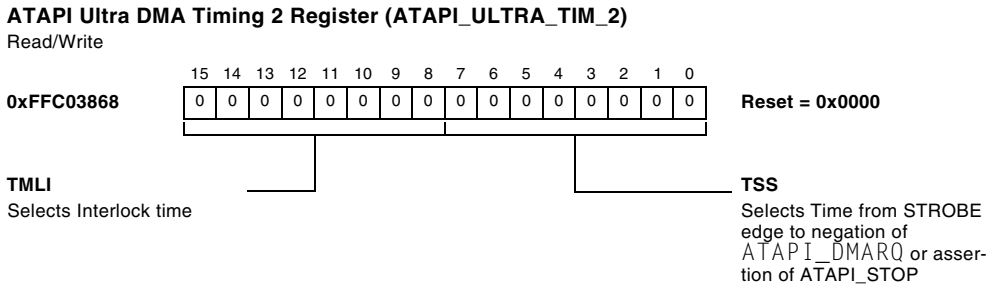


Figure 21-46. ATAPI Ultra DMA Timing 2 Register

ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3) Register

The ATAPI_ULTRA_TIM_3 register (see Figure 21-47) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

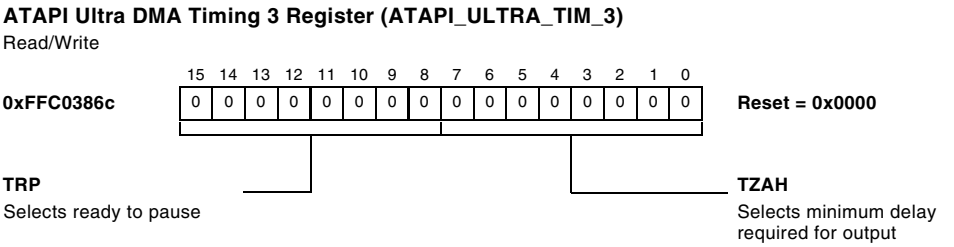


Figure 21-47. ATAPI Ultra DMA Timing 3 Register

Ultra DMA mode 5 can be used only when SCLK = 133 MHz. Ultra DMA mode 4 requires SCLK = 100 MHz and above. The other Ultra DMA modes can used at SCLK frequencies lower than 100 MHz.

ATAPI Device I/O Registers

These are the registers present in an ATAPI-compliant device.

Table 21-7 shows a list of ATAPI device I/O registers present on ATAPI-compliant devices.

Table 21-7. ATAPI Device I/O Registers

ATAPI_CS1-0	ADDR3-1	READ (ATAPI_DIOR)	WRITE (ATAPI_DIOW)
NN	XXX	Data Bus (Z)	Not Used
Control Block Registers			
AN	0XX	Data Bus (Z)	Not Used
AN	10X	Data Bus (Z)	Not Used
AN	110 (0x0E)	Alternate Status	Device Control
AN	111	Not Used	Not Used
Command Block Registers			
NA	000 (0x00)	PIO Data	PIO Data
NA	001 (0x01)	Error	Feature
NA	010 (0x02)	Sector Count	Sector Count
NA	011 (0x03)	LBA (low 0-7)	LBA (low 0-7)
NA	100 (0x04)	LBA (mid 8-15)	LBA (mid 8-15)
NA	101 (0x05)	LBA (high 16-23)	LBA (high16-23)
NA	110 (0x06)	Device	Device
NA	111 (0x07)	Status	Command

(A: Asserted N: Negated)

The ATAPI I/O registers are all accessed using PIO transfers. When an access is made to an 8-bit register, the data is expected on ATAPI_D7-0 for a write access and presented on ATAPI_D7-0 for a read access. When an access is made to a 16-bit register, the data is expected on ATAPI_D15-0 for a write access and presented on ATAPI_D15-0 for a read access.

The ATAPI I/O registers are addressed using the $\overline{\text{ATAPI\_CS1-0}}$ and ADDR3-1 lines. These lines are mapped into the core's address range using the DEV_ADDR register of the ATAPI host, making them transparent for any software wanting to access them. The registers can be mapped into the 0x00 to 0x0F address range, according to the following scheme.

- $\overline{\text{ATAPI\_CS0}}$ <= ADR_I (3)
- $\overline{\text{ATAPI\_CS1}}$ <= not ADR_I (3)
- ADDR2-0 <= ADR_I (2:0)
- $\overline{\text{ATAPI\_CS1-0}}$ reflect the ADR_(3) signal state.
- $\overline{\text{ATAPI\_CS0}}$ is asserted (low level) when ADR_I (3) is negated ('0').
- $\overline{\text{ATAPI\_CS1}}$ is asserted (low level) when ADR_I (3) is asserted ('1').
- ADDR3-1 reflects the ADR_I (2:0) state.

This makes Device Address map as follows:

- 0x00: Device PIO Data Port/DMA Data Port/Ultra DMA Port
- 0x01 – 0x07: Device Command Block Registers
- 0x08 -- 0x0F: Device Control Block Registers

The various device registers addressable are detailed as follows.

Command Register (R/W)

The command register contains the command code being sent to the device. command execution begins immediately after this register is written. The contents of the command block registers become parameters of the command when this register is written. Writing this register clears any pending interrupt condition. For all commands except DEVICE RESET, this register shall only be written when BSY and DRQ are both cleared to zero and $\overline{\text{ATAPI\_DMACK}}$ is not asserted.

Device Control Register (WO)

The device control register allows a host to perform a software reset of attached devices and to enable or disable the assertion of the INTRQ signal by a selected device. It contains Software Reset (SRST), Interrupt Enable (nIEN), and High Order Byte (HOB) bits for the 48-bit address feature set, as shown in Figure 21-48. When the Device Control register is written, both devices respond to the write regardless of which device is selected. When the SRST bit is set to 1, both devices shall perform the software reset protocol. This register contains software reset, Interrupt Enable & High Order Byte bits for 48-bit address feature set.

7	6	5	4	3	2	1	0
HOB	r	r	r	r	SRST	nIEN	0

Figure 21-48. Device Control Register

Bit 1: nIEN: If nIEN is set, the device should release INTRQ. If it is clear, INTRQ should be enabled.

Features Register (WO)

The contents of this register becomes a command parameter after the command is written and the meaning of this parameter is command dependent.

Sector Count Register (R/W)

The sector count register holds the number of sectors to be read or written.

Status Register (RO)

The status register contains the device status. The register's contents are updated to reflect the current state of the device and the progress of any command being executed by the device. Reading the status register clears any pending interrupt. The host should not read the status register when an interrupt is expected as this may clear the interrupt pending before the ATAPI_INTRQ can be recognized. The host should generally read the alternate status register to prevent unwanted clearing of pending interrupts. When INTRQ is asserted, the host can read the Status register to know the current status.

7	6	5	4	3	2	1	0
BSY	DRDY	#	#	DRQ	obsolete	obsolete	ERR

Figure 21-49. Status Register

Bit 7: BSY bit is set by the device during the following events:

- After a command is written (if DRQ is not set).
- Between blocks of data transfer during PIO data-in (before DRQ is cleared).
- After transfer of data block during PIO data-out (before DRQ is cleared).
- During data transfer of DMA commands:
    - If BSY = 1, device is in control of status register
    - If BSY = 0, host is in control of status register

Alternate Status Register (RO)

The alternate status register contains the same information as the status register, but a pending interrupt is not cleared when this register is read.

Error Register (RO)

The error register contents are valid, when ERR bit in the status register is set (BSY = 0 & DRQ = 0) at the end of command completion (except EXECUTE DEVICE DIAGNOSTICS or DEVICE RESET).

The register contains a diagnostic code following a power-on, hardware or software reset or command completion of EXECUTE DEVICE DIAGNOSTIC OR DEVICE RESET.

Bit 2: ABORT: Says that the particular command is not supported.

All other bit commands are dependent.

ATAPI Standards Reference

The following ATA standards contribute to the ATAPI standard. Refer to the ATAPI specification for full details. In addition to these terms, this reference section provides:

- "Summary of IDE/ATA Standards" on page 21-77
- "ATAPI Timing Summary" on page 21-78
- "IDE/ATA Transfer Modes and Protocols" on page 21-78
- "ATAPI Device Selection" on page 21-80

ATA (ATA-1)

The original IDE/ATA standard defines the following features and transfer modes:

- Two Hard Disks: The specification calls for a single channel in a PC, shared by two devices that are configured as master and slave.
- PIO Modes: ATA includes support for PIO modes 0, 1 and 2.
- DMA Modes: ATA includes support for single word DMA modes 0, 1 and 2, and multiword DMA mode 0.

ATA-2

ATA-2 was a significant enhancement of the original ATA standard. It defines the following improvements over the base ATA standard (with which it is backward compatible):

- Faster PIO Modes: ATA-2 adds the faster PIO modes 3 and 4 to those supported by ATA.
- Faster DMA Modes: ATA-2 adds multiword DMA modes 1 and 2 to the ATA modes.
- Block Transfers: ATA-2 adds commands to allow block transfers for improved performance.
- Logical Block Addressing (LBA): ATA-2 defines support (by the hard disk) for logical block addressing. Using LBA requires BIOS support on the other end of the interface as well.
- Improved Identify Drive Command: This command allows hard disks to respond to inquiries from software, with more accurate information about their geometry and other characteristics.

ATA-3

The ATA-3 standard is a minor revision of ATA-2, which was published in 1997 as ANSI standard X3.298-1997, *AT Attachment 3 Interface*. It defines the following improvements compared to ATA-2 (with which it is backward compatible):

- Improved Reliability: ATA-3 improves the reliability of the higher-speed transfer modes, which can be an issue due to the low-performance standard cable used up to that point in IDE/ATA. (An improved cable was defined as part of ATA/ATAPI-4.)
- Self-Monitoring Analysis and Reporting Technology (SMART): ATA-3 introduced this reliability feature.
- Security Feature: ATA-3 defined security mode, which allows devices to be protected with a password.

ATA/ATAPI-4

- Ultra DMA Modes: High-speed Ultra DMA modes 0, 1 and 2, defining transfer rates of 16.7, 25 and 33.3 MB/s were created.
- High-Performance IDE Cable: An improved, 80-conductor IDE cable was first defined in this standard. It was thought that the higher-speed Ultra DMA modes would require the use of this cable in order to eliminate interference caused by their higher speed. In the end, the use of this cable was left "optional" for these modes. (It became mandatory under the still faster Ultra DMA modes defined in ATA/ATAPI-5.)
- Cyclical Redundancy Checking (CRC): This feature was added to ensure the integrity of data sent using the faster Ultra DMA modes.

- Advanced Commands Defined: Special command queuing and overlapping protocols were defined.
- Command Removal: The command set was “cleaned up”, with several older, obsolete commands removed.

ATA/ATAPI-5

The changes defined in ATA/ATAPI-5 include:

- New Ultra DMA Modes: Higher-speed Ultra DMA modes 3 and 4, defining transfer rates of 44.4 and 66.7 MB/s were specified.
- Mandatory 80-Conductor IDE Cable Use: The improved 80-conductor IDE cable first defined in ATA/ATAPI-4 for optional use is made mandatory for Ultra DMA modes 3 and 4. ATA/ATAPI-5 also defines a method by which a host system can detect if an 80-conductor cable is in use, so it can determine whether or not to enable the higher speed transfer modes.
- Miscellaneous Command Changes: A few interface commands were changed, and some old ones deleted.

ATA/ATAPI-6

- New Ultra DMA Modes: Higher-speed Ultra DMA mode 5, defining a transfer rate of 100 MB/s was specified.
- Mandatory LBA Mode Usage: CHS mode operation not supported.

Summary of IDE/ATA Standards

Table 21-8. IDE/ATA Standards

Interface Standard	ANSI Standard Number (includes date)	PIO Modes Added	DMA Modes Added	Ultra DMA Modes Added	Notable Features or Enhancements Introduced
ATA-1	X3.221-1994	0, 1, 2	Single word 0, 1, 2; multiword 0	--	--
ATA-2	X3.279-1996	3, 4	Multiword 1, 2	--	Block transfers, Logical block addressing, Improved identify drive command
ATA-3	X3.298-1997	--	--	--	Improved reliability, SMART, Drive security
ATA ATAPI-4	NCITS 317-1998	--	--	0, 1, 2	Ultra DMA, 80-conductor IDE cable, CRC
ATA ATAPI-5	NCITS 340-2000	--	--	3, 4	--
ATA ATAPI-6		--	--	5	LBA expansion, Acoustic management, Multimedia streaming

ATAPI Timing Summary

The timings mentioned below are the minimum timings. The maximum timing is dependent on the devices and is usually using the ACK signal.

- Ultra DMA (M5, M4, M3, M2, M1, M0)- 40, 60, 90, 120, 160, 240 ns
- Multi DMA (M2, M1. M0)- 120, 150, 480 ns
- PIO Access (M4, M3, M2, M1, M0)- 120, 180, 240, 383, 600 ns

IDE/ATA Transfer Modes and Protocols

The following sections describe IDE/ATA transfer modes and protocols.

Programmed (I/O) PIO Modes

Table 21-9. Programmed I/O Modes

PIO Mode	Cycle Time (ns)	Maximum Transfer Rate (MB/s)	Defining Standard
Mode 0	600	3.3	ATA
Mode 1	383	5.2	ATA
Mode 2	240	8.3	ATA
Mode 3	180	11.1	ATA-2
Mode 4	120	16.7	ATA-2

The maximum transfer rate is double the reciprocal of the cycle time, doubled because the IDE/ATA interface is two bytes (16 bits) wide.

Direct Memory Access (DMA) Modes

Table 21-10. Multiword DMA Modes

DMA Mode	Cycle Time (ns)	Maximum Transfer Rate (MB/s)	Defining Standard
Multiword Mode 0	480	4.2	ATA
Multiword Mode 1	150	13.3	ATA-2
Multiword Mode 2	120	16.7	ATA-2

Ultra Direct Memory Access (DMA) Modes

The first implementation of Ultra DMA was specified in the ATA/ATAPI-4 standard and included three Ultra DMA modes, providing up to 33 MB/s of throughput. Several newer, faster Ultra DMA modes were added in subsequent years. The table shows all of the current Ultra DMA modes, along with their cycle times and maximum transfer rates.

Table 21-11. Ultra DMA Modes

Ultra DMA Mode	Cycle Time (ns)	Maximum Transfer Rate (MB/s)	Defining Standard
Mode 0	240	16.7	ATA/ATAPI-4
Mode 1	160	25.0	ATA/ATAPI-4
Mode 2	120	33.3	ATA/ATAPI-4
Mode 3	90	44.4	ATA/ATAPI-5
Mode 4	60	66.7	ATA/ATAPI-5
Mode 5	40	100.0	ATA/ATAPI-6

The cycle time shows the speed of the interface clock. Double transition clocking is what allows Ultra DMA mode 2 to have a maximum transfer rate of 33.3 MB/s despite having a clock cycle time identical to “regular DMA” multiword mode 2, which has half that maximum.

Even with the advantage of double transition clocking, going above 33 MB/s finally exceeded the capabilities of the old 40-conductor standard IDE cable. To use Ultra DMA modes over 2, a special, 80-conductor IDE cable is required. This cable uses the same 40 pins as the old cables, but adds 40 ground lines between the original 40 signals to separate those lines from each other and prevent interference and data corruption. (The 80-conductor cable was actually specified in ATA/ATAPI-4 along with the first Ultra DMA modes, but it was “optional” for modes 0, 1 and 2.)

ATAPI Device Selection

DEV0: CSEL is negated.

If the CSEL (cable select) of the device is connected to the CSEL of the cable and ground, the device recognizes itself as DEV0.

DEV1: CSEL is asserted.

If the CSEL of the device is not connected, it recognizes as DEV1

The host discriminates the two devices by writing the DEV bit in device register. When two devices are connected on the cable, commands are written in parallel to both devices. For all commands except EXECUTE DEVICE DIAGNOSTICS, only the selected device executes the command. Both devices shall execute an EXECUTE DEVICE DIAGNOSTIC regardless of which device is selected and DEV1 will post status to DEV0 through $\overline{\text{ATAPI\_PDIAG}}$.

When the DEV bit is set to 0, DEV0 is selected. When the DEV bit is set to 1, DEV1 is selected

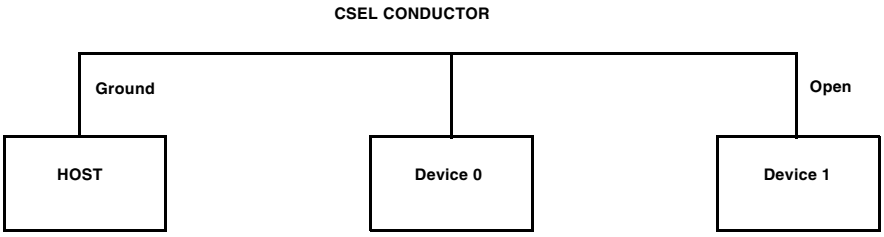


Figure 21-50. ATAPI Device Selection

22 SPI-COMPATIBLE PORT CONTROLLERS

This chapter describes the serial peripheral interface (SPI) ports and includes the following sections:

- "Overview" on page 22-1
- "Interface Overview" on page 22-3
- "Description of Operation" on page 22-16
- "Functional Description" on page 22-26
- "Programming Model" on page 22-30
- "SPI Registers" on page 22-43
- "Programming Examples" on page 22-49

Overview

The processor has up to three SPI ports that provide an I/O interface to a wide variety of SPI-compatible peripheral devices.

With a range of configurable options, the SPI ports provide a glueless hardware interface with other SPI-compatible devices. 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.

SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. Table 22-1 lists the critical SPI signals.

Table 22-1. SPI Signals

Signal Name	Function
SPIxSCK	SPI Clock Signal Pin
SPIxMOSI	Master Out Slave In Data Pin
SPIxMISO	Master In Slave Out Data Pin
$\overline{\text{SPIxSS}}$	SPI Device-Select Input Pin

Each 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
- 3 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

Interface Overview

Figure 22-1 provides a block diagram of each SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SPIxSCK 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 SPIxSCK synchronizes the shifting and sampling of the data on the two serial data pins.

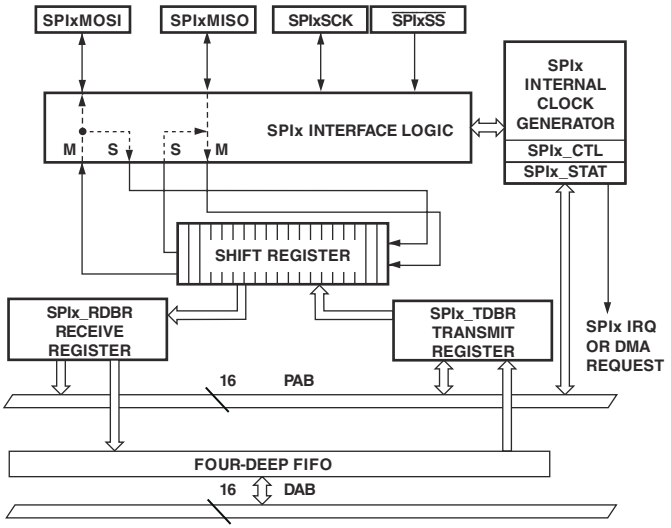


Figure 22-1. SPIx Block Diagram

External Interface

All of the SPI signals are accessible through GPIO ports. The four SPI signals that make up the 4-wire interface (SPIxSCK, SPIxMISO, SPIxMOSI, and $\overline{\text{SPIxSS}}$) are sometimes multiplexed with other peripherals. By default, all pins function as GPIOs and each can be individually enabled to function as an SPI pin by the respective bits in the appropriate PORTx_FER register.

If the configurable pin is shared among multiple peripherals, the associated PORTx_MUX register will also need to be written to explicitly configure it as SPI. Table 22-2 provides a mapping of SPI pins to GPIO pins along with an explanation regarding how to enable these pins for use as SPI.

Table 22-2. SPI/GPIO Pin Mapping and Programming Instructions

SPI Signal	GPIO Pin	To configure for SPI use
SPI0SCK	PE0	PORTE_FER[0] = 1, PORTE_MUX[1:0] = b#00
SPI0MOSI	PE2	PORTE_FER[2] = 1, PORTE_MUX[5:4] = b#00
SPI0MISO	PE1	PORTE_FER[1] = 1, PORTE_MUX[3:2] = b#00
$\overline{\text{SPI0SS}}$	PE3	PORTE_FER[3] = 1, PORTE_MUX[7:6] = b#00
SPI1SCK	PG8	PORTG_FER[8] = 1, PORTG_MUX[17:16] = b#00
SPI1MOSI	PG10	PORTG_FER[10] = 1, PORTG_MUX[21:20] = b#00
SPI1MISO	PG9	PORTG_FER[9] = 1, PORTG_MUX[19:18] = b#00
$\overline{\text{SPI1SS}}$	PG11	PORTG_FER[11] = 1, PORTG_MUX[23:22] = b#00
SPI2SCK	PB12	PORTB_FER[12] = 1, PORTB_MUX[25:24] = b#00
SPI2MOSI	PB13	PORTB_FER[13] = 1, PORTB_MUX[27:26] = b#00
SPI2MISO	PB14	PORTB_FER[14] = 1, PORTB_MUX[29:28] = b#00
$\overline{\text{SPI2SS}}$	PB8	PORTB_FER[8] = 1, PORTB_MUX[17:16] = b#00

Each SPI also features three slave select output signals that are sometimes multiplexed with other peripheral signals. They can be enabled on an individual basis using the PORTx_FER and PORTx_MUX registers. Again, the pins are enabled as GPIO by default. Table 9-3 on page 9-9 provides a mapping of SPI pins to GPIO pins along with an explanation regarding how to enable these pins for use as SPI. For more information see Chapter 9, "General-Purpose Ports".

Serial Peripheral Interface Clock Signal (SPIxSCK)

The SPIxSCK 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 SPIxSCK 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 SPIxSCK 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 $\overline{\text{SPIxSS}}$ input is driven inactive (high).

The SPIxSCK is used to shift out and shift in the data driven on the SPIxMISO and SPIxMOSI lines, see "SPI Transfer Protocols" on page 22-17. Clock polarity and clock phase relative to data are programmable in the SPIx_CTL register and define the transfer format.

The SPIxSCK signal is routed to a shared port 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 appropriate PORTx_FER register to enable the pin for peripheral use, and then verify that the associated PORTx_MUX register is properly set to specifically enable the SPI clock functionality. For more information see Chapter 9, "General-Purpose Ports".

Master Out Slave In (MOSI)

The SPIxMOSI 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 SPIxMOSI pin becomes a data transmit (output) pin, transmitting output data. If the processor is configured as a slave, the SPIxMOSI pin becomes a data receive (input) pin, receiving input data. In an SPI interconnection, the data is shifted out from the SPIxMOSI output pin of the master and shifted into the SPIxMOSI input(s) of the slave(s).

The SPIxMOSI signal is routed to a shared port pin which functions as a GPIO by default. To enable this pin for use as the SPI SPIxMOSI signal, be sure to first configure the appropriate PORTx_FER register to enable the pin for peripheral use, and then verify that the associated PORTx_MUX register is properly set to specifically enable the SPI Master Out Slave In functionality. For more information see Chapter 9, “General-Purpose Ports”.

Master In Slave Out (MISO)

The SPIxMISO 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 SPIxMISO pin becomes a data receive (input) pin, receiving input data. If the processor is configured as a slave, the SPIxMISO pin becomes a data transmit (output) pin, transmitting output data. In an SPI interconnection, the data is shifted out from the SPIxMISO output pin of the slave and shifted into the SPIxMISO input pin of the master.

The SPIxMISO signal is routed to a shared port pin which functions as a GPIO by default. To enable this pin for use as the SPIxMISO signal, be sure to first configure the appropriate PORTx_FER register to enable the pin for peripheral use, and then verify that the associated PORTx_MUX register is properly set to specifically enable the SPI Master In Slave Out functionality. For more information see Chapter 9, “General-Purpose Ports”.

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 22-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host micro controller is the SPI master.

The processor can be booted by way of its SPI interface to allow user application code and data to be downloaded before runtime.

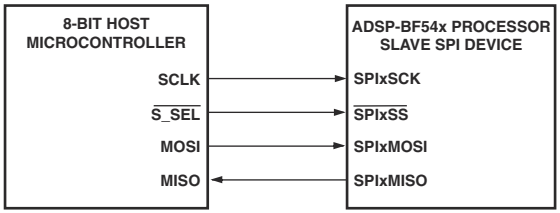


Figure 22-2. ADSP-BF54x processor as Slave SPI Device

Serial Peripheral Interface Slave Select Input Signal

The $\overline{\texttt{SPIxSS}}$ 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 $\overline{\texttt{SPIxSS}}$ input signal of a master is asserted (driven low), and the PSSE bit in the SPIx_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The $\overline{\texttt{SPIxSS}}$ signal is routed to a shared port 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 appropriate PORTx_FER register is properly set to enable the pin for peripheral use, and then verify that the

associated PORTx_MUX register is set to specifically enable the SPI slave select input functionality. For more information see Chapter 9, “General-Purpose Ports”.

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 SPIxSCK period. See Figure 22-3. The minimum time between successive word transfers (T4) is two SPIxSCK periods. This is measured from the last active edge of SPIxSCK of one word to the first active edge of SPIxSCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

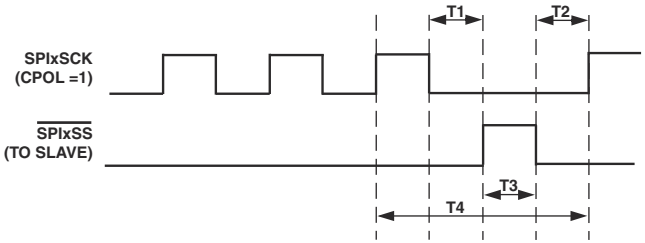


Figure 22-3. SPI Timing

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SPIxSCK period. In this case, T1 and T2 will each always be equal to one-half the SPIxSCK 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 three slave select enable signals automatically. See “SPIx Flag Register” on page 22-46 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 22-3 summarizes how to setup the port control logic in order to enable the individual slave select enable outputs.

Table 22-3. SPI Slave Select Enable Setup

Signal Name	Pin Name	Port Control To Enable Signal
$\overline{\text{SPI0SEL1}}$	PE4	Set bit 4 in PORTE_FER = 1 Set PORTE_MUX[9:8] = b#00
$\overline{\text{SPI0SEL2}}$	PE5	Set bit 5 in PORTE_FER = 1 Set PORTE_MUX[11:10] = b#00
$\overline{\text{SPI0SEL3}}$	PE6	Set bit 6 in PORTE_FER = 1 Set PORTE_MUX[13:12] = b#00
$\overline{\text{SPI1SEL1}}$	PG5	Set bit 5 in PORTG_FER = 1 Set PORTG_MUX[11:10] = b#00
$\overline{\text{SPI1SEL2}}$	PG6	Set bit 6 in PORTG_FER = 1 Set PORTG_MUX[13:12] = b#00
$\overline{\text{SPI1SEL3}}$	PG7	Set bit 7 in PORTG_FER = 1 Set PORTG_MUX[15:14] = b#00
$\overline{\text{SPI2SEL1}}$	PB9	Set bit 9 in PORTB_FER = 1 Set PORTB_MUX[19:18] = b#00
$\overline{\text{SPI2SEL2}}$	PB10	Set bit 10 in PORTB_FER = 1 Set PORTB_MUX[21:20] = b#00
$\overline{\text{SPI2SEL3}}$	PB11	Set bit 11 in PORTB_FER = 1 Set PORTB_MUX[23:22] = b#00

If enabled as a master, each SPI uses its SPIx_FLG register to enable up to three general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the PBx, PEx, and PGx 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 PBx, PEx, and PGx pins must be enabled for use by SPI in the appropriate PORTx_FER and PORTx_MUX registers. For more information see Chapter 9, “General-Purpose Ports”.

Refer to Table 22-4 for more details regarding which port pins must be configured prior to being modified by way of the SPIx_FLG register.

In slave mode, the SPIx_FLG bits have no effect, and each SPI uses the $\overline{\text{SPIxSS}}$ input as a slave select. Just as in the master mode case, the $\overline{\text{SPIxSS}}$ pin must first be configured as a peripheral pin in the PORTx_MUX register, and then as an SPI pin in the PORTx_FER register. Figure 22-14 on page 22-46 shows the SPIx_FLG register diagram.

Table 22-4. SPIx_FLG Bit Mapping to Port Pins

Bit	Name	Function	Port Pin	Default
0		Reserved		0
1	FLS1	$\overline{\text{SPIxSEL1}}$ Enable	SPI0: PE4 SPI1: PG5 SPI2: PB9	0
2	FLS2	$\overline{\text{SPIxSEL2}}$ Enable	SPI0: PE5 SPI1: PG6 SPI2: PB10	0
3	FLS3	$\overline{\text{SPIxSEL3}}$ Enable	SPI0: PE6 SPI1: PG7 SPI2: PB11	0
4		Reserved		0
5		Reserved		0
6		Reserved		0
7		Reserved		0
8		Reserved		1
9	FLG1	$\overline{\text{SPIxSEL1}}$ Value	SPI0: PE4 SPI1: PG5 SPI2: PB9	1
10	FLG2	$\overline{\text{SPIxSEL2}}$ Value	SPI0: PE5 SPI1: PG6 SPI2: PB10	1
11	FLG3	$\overline{\text{SPIxSEL3}}$ Value	SPI0: PE6 SPI1: PG7 SPI2: PB11	1
12		Reserved		1
13		Reserved		1
14		Reserved		1
15		Reserved		1

Slave Select Inputs

If the SPI is in slave mode, $\overline{\text{SPIxSS}}$ acts as the slave select input. When enabled as a master, $\overline{\text{SPIxSS}}$ can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPIx_CTL enables this feature. When PSSE = 1, the $\overline{\text{SPIxSS}}$ input is the master mode error input. Otherwise, $\overline{\text{SPIxSS}}$ is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPIx_FLG register are used in a multiple slave SPI environment. For example, if there are four SPI devices in the system including a processor master, the master processor can support the SPI mode transactions across the other three devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The three 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 $\overline{\text{SPIxSS}}$ pins. In this configuration, the FLSx bits in SPIx_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the three microcontrollers/peripherals with SPI interfaces are slaves. In case 3, all four devices connected by way of SPI ports can be other processors.

1. Transmit to all three 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.

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 two other slave processors) at a time and transmit broadcast data to all three 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 22-4 shows one processor as a master with three other SPI-compatible devices as slaves.

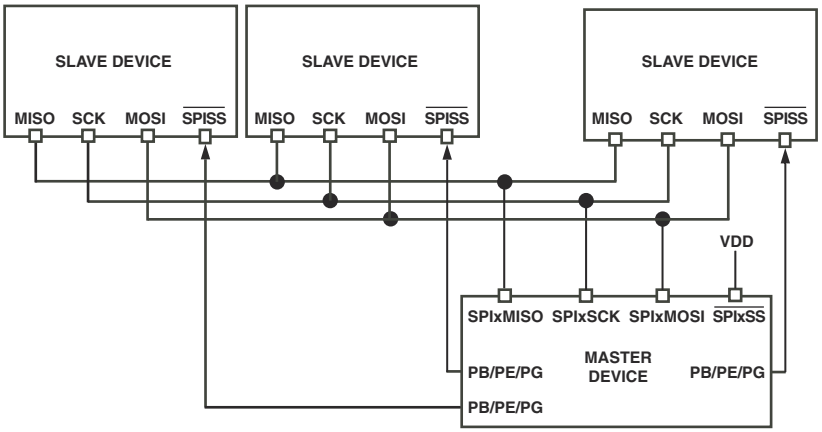


Figure 22-4. Single-Master, Multiple-Slave Configuration

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 SPIx_STAT register until it goes high for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word is transferred.

Internal Interfaces

Each 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. For more information on the default arbitration priority see Chapter 2, "Chip Bus Hierarchy".

DMA Functionality

Each SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data 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 SPIx_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently goes high, the last word is transferred.

The four-word FIFO is cleared when the SPI port is disabled.

SPI Transmit Data Buffer

The SPIx_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 SPIx_TDBR is loaded into the internal shift register, which is inaccessible by software. A read of SPIx_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into the SPIx_TDBR register for transmission just prior to the beginning of a data transfer. A write to SPIx_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPIx_TDBR are repeatedly transmitted. A write to SPIx_TDBR is permitted in this mode, and this data is transmitted.

If the SZ control bit in the SPIx_CTL register is set, SPIx_TDBR may be reset to 0 under certain circumstances.

If multiple writes to SPIx_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPIx_TDBR are transmitted. Multiple writes to SPIx_TDBR are possible, but not recommended.

SPI Receive Data Buffer

The SPIx_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 SPIx_RDBR. During a DMA receive operation, the data in SPIx_RDBR is automatically read by the DMA. When SPIx_RDBR is read by way of software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD = b#00).

The SPIx_SHADOW register is provided for use in debugging software. This register is at a different address than the receive data buffer, SPIx_RDBR, but its contents are identical to that of SPIx_RDBR. When a software read of SPIx_RDBR occurs, the RXS bit in SPIx_STAT is cleared and an SPI transfer may be initiated (if TIMOD = b#00 in SPIx_CTL). No such hardware action occurs when the SPIx_SHADOW register is read. The SPIx_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 (SPI modes 0-3). These combinations are selected using the CPOL and CPHA bits in SPIx_CTL, as shown in Figure 22-5.

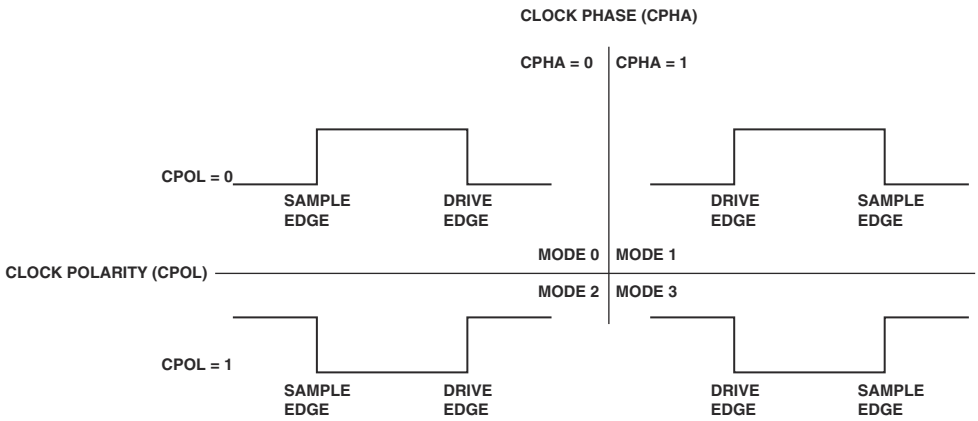


Figure 22-5. SPI Modes of Operation

The figures "SPI Transfer Protocol for CPHA = 0" on page 22-18 and "SPI Transfer Protocol for CPHA = 1" on page 22-19 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SPIxSCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SPIxSCK, SPIxMISO, and SPIxMOSI pins are directly connected between the master and the slave. The SPIxMISO signal is the output from the slave (slave transmission), and the SPIxMOSI signal is the output from the master (master transmission). The SPIxSCK signal is generated by the master, and the $\overline{\text{SPIxSS}}$ 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 SPIx_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, $\overline{\text{SPIxSS}}$, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, $\overline{\text{SPIxSS}}$ may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software by way of manipulation of SPIx_FLG.

Figure 22-6 shows the SPI transfer protocol for CPHA = 0. Note SPIxSCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

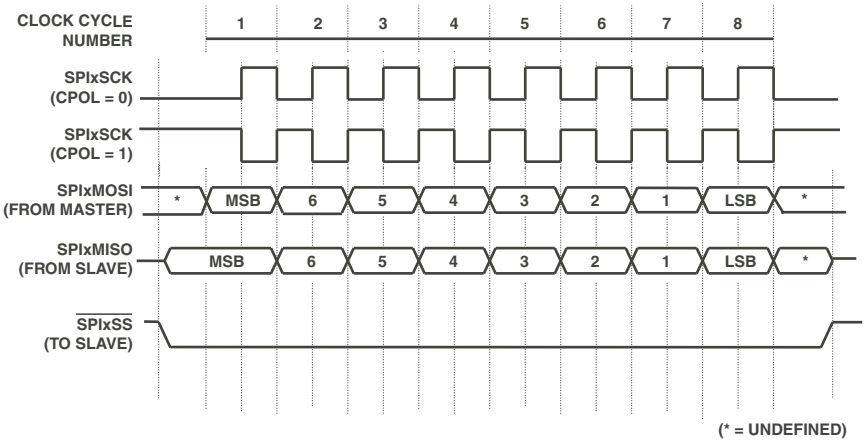


Figure 22-6. SPI Transfer Protocol for CPHA = 0

Figure 22-7 shows the SPI transfer protocol for CPHA = 1. Note SPIxSCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

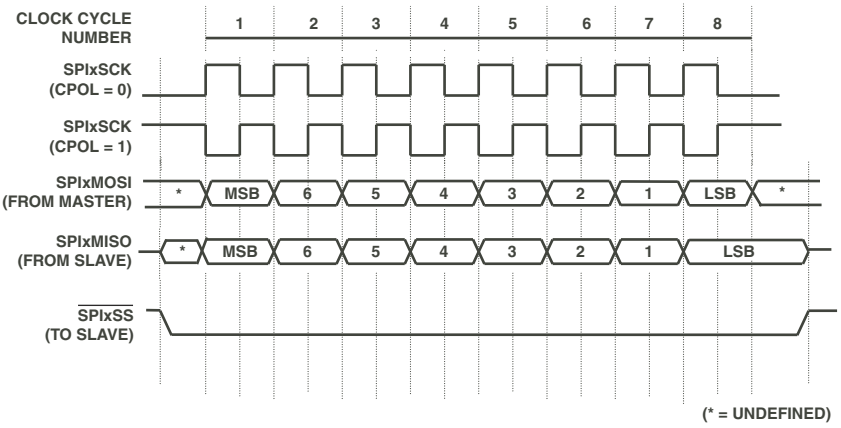


Figure 22-7. SPI Transfer Protocol for CPHA = 1

SPI General Operation

Each SPI can be used in a single master as well as multimaster environment. The SPIxMOSI, SPIxMISO, and the SPIxSCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode is selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the SPIxMISO 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 (SPIxMOSI and SPIxMISO) 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 SPIxMOSI and SPIxMISO 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 SPIxMOSI pin is three-stated when the data driven out on SPIxMOSI is a logic high. The SPIxMOSI 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 SPIxMISO pin is three-stated if the data driven out on SPIxMISO 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 ($\overline{\text{SPIxSS}}$). 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 by way of their SPI ports, all SPIxMOSI pins are connected together, all SPIxMISO pins are connected together, and all SPIxSCK pins are connected together.

For a multislave environment, the processor can make use of three programmable flags for each SPI port, which are dedicated SPI slave select signals for the SPI slave devices. See Table 22-4 on page 22-11.

At reset, the SPI is disabled and configured as a slave.

SPI Control

The SPIx_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 SPIx_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to b#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 b#01, the transaction is initiated when the transmit buffer is written. A value of b#10 selects DMA receive mode and the first transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPIx_RDBR. A value of b#11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPIx_TDBR.

The PSSE bit is used to enable the $\overline{\text{SPIxSS}}$ input for master. When not used, $\overline{\text{SPIxSS}}$ can be disabled, freeing up a chip pin as general-purpose I/O.

The EMISO bit enables the SPIxMISO 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 SPIx_STAT register is set. See “Mode Fault Error (MODF)” on page 22-24.

Figure 22-13 on page 22-45 provides the bit descriptions for SPIx_CTL.

Clock Signals

The SPIxSCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value of SPIx_BAUD. For slave devices, the value in SPIx_BAUD is ignored. When the SPI device is a master, SPIxSCK is an output signal. When the SPI is a slave, SPIxSCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high). See Figure 22-5 on page 22-17.

The SPIxSCK signal is used to shift out and shift in the data driven onto the MSPIxMISO and SPIxMOSI lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and clock phase relative to data are programmable into SPIx_CTL and define the transfer format.

SPI Baud Rate

The SPIx_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:

$$SPIxSCK\ Frequency = \frac{SCLK\ (System\ Clock\ Frequency)}{2 \times SPIx\_BAUD}$$

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

Table 22-5 lists several possible baud rate values for SPIx_BAUD.

Table 22-5. SPI Master Baud Rate Example

SPIx_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 SPIx_STAT register is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPIx_STAT register can be read at any time.

Some of the bits in SPIx_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a 1 to the desired bit position of SPIx_STAT. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPIx_STAT to clear the TXE error condition. This allows the user to read SPIx_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

See Figure 22-15 on page 22-48 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPIx_STAT when the $\overline{\text{SPIxSS}}$ input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPIx_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPIx_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPIx_CTL is cleared, disabling the SPI system
- The MODF status bit in SPIx_STAT is set
- An SPI error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the $\overline{\text{SPIxSS}}$ input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another mode fault error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (SPIx MOSI, SPIxMISO, and SPIxSCK) 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 SPIx_STAT when all the conditions of transmission are met, and there is no new data in SPIx_TDBR (SPIx_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPIx_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPIx_STAT register when a new transfer is completed, but before the previous data can be read from SPIx_RDBR. The state of the GM bit in the SPIx_CTL register determines whether SPIx_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPIx_STAT when a write to SPIx_TDBR coincides with the load of the shift register. The write to SPIx_TDBR can be by way of software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPIx_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

Interrupt Output

Each SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the TIMOD field in the SPIx_CTL register. In DMA mode (TIMOD = b#1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = b#11) or read from (TIMOD = b#10). In non-DMA mode (TIMOD = b#0X), a data interrupt is generated when the SPIx_TDBR is ready to be written to (TIMOD = b#01) or when the SPIx_RDBR is ready to be read from (TIMOD = b#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 = b#11) or an overflow (RBSY when TIMOD = b#10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPIx_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control" on page 22-21.

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 PORTx_FER and/or PORTx_MUX registers to properly configure the required PBx, PEx, and/or PGx pins for SPI use.
2. The core writes to SPIx_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 SPIx_BAUD and SPIx_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 SPIx_FLG.
5. The TIMOD bits in SPIx_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 (SPIx_TDBR) or a data read of the receive data buffer (SPIx_RDBR).
6. The SPI then generates the programmed clock pulses on SPIxSCK and simultaneously shifts data out of SPIxMOSI and shifts data in from SPIxMISO. Before a shift, the shift register is loaded with the contents of the SPIx_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPIx_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 22-8 on page 22-39 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 SPIx_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the SPIxMOSI 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 SPIxMISO pin, overwriting the older data in the SPIx_RDBR buffer. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPIx_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 SPIx_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of SPIx_RDBR or a write to SPIx_TDBR. This is summarized in Table 22-6.

If the SPI port is enabled with TIMOD = b#01 or TIMOD = b#11, the hardware immediately issues a first interrupt or DMA request.

Table 22-6. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
b#00	Transmit and Receive	Initiate new single word transfer upon read of SPIx_RDBR and previous transfer completed.	Interrupt active when receive buffer is full. Read of SPIx_RDBR clears interrupt.
b#01	Transmit and Receive	Initiate new single word transfer upon write to SPIx_TDBR and previous transfer completed.	Interrupt active when transmit buffer is empty. Writing to SPIx_TDBR clears interrupt.
b#10	Receive with DMA	Initiate new multiword transfer upon enabling SPIx for DMA mode. Individual word transfers begin with a DMA read of SPIx_RDBR, and last transfer completed.	Request DMA reads as long as SPIx DMA FIFO is not empty.
b#11	Transmit with DMA	Initiate new multiword transfer upon enabling SPIx for DMA mode. Individual word transfers begin with a DMA write to SPIx_TDBR, and last transfer completed.	Request DMA writes as long as SPIx 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 $\overline{\text{SPIxSS}}$ select signal to the active state (low), or by the first active edge of the clock (SPIxSCK), depending on the state of CPHA.

These steps illustrate SPI operation in the slave mode:

1. The core writes to the appropriate PORTx_FER and PORTx_MUX registers to properly configure the GPIO pins as SPI signals.
2. The core writes to SPIx_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.
3. To prepare for the data transfer, the core writes data to be transmitted into SPIx_TDBR.
4. Once the $\overline{\text{SPIxSS}}$ falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCK edges, depending on the states of CPHA and CPOL.
5. Reception/transmission continues until $\overline{\text{SPIxSS}}$ 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 $\overline{\text{SPIxSS}}$ and/or SPIxSCK clock edge.

See Figure 22-8 on page 22-39 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 SPIx_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the SPIxMISO 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 SPIxMOSI pin, overwriting the older data in SPIx_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPIx_RDBR is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 22-7 are necessary to prepare the device for a new transfer.

Table 22-7. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and receive	Interrupt active when receive buffer is full. Read of SPIx_RDBR clears interrupt.
b#01	Transmit and receive	Interrupt active when transmit buffer is empty. Writing to SPIx_TDBR clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as SPIx DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as SPIx 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 SPIx_TDBR is written to or SPIx_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SPIxSCK signal remains inactive for the first half of the first cycle of SPIxSCK. For a slave with CPHA = 0, the transfer starts as soon as the $\overline{\text{SPIxSS}}$ input goes low.

For CPHA = 1, a transfer starts with the first active edge of SPIxSCK 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 SPIxSCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPIx_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SPIxSCK. For a slave SPI, RXS is set shortly after the last SPIxSCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPIx_RDBR is full (TIMOD = b#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 ($\overline{\text{SPIxSS}}$ going low for CPHA = 0, first active edge of SPIxSCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a

transfer (either by writing the SPIx_TDBR or reading the SPIx_RDBR, depending on TIMOD), and is set one-half SPIxSCK period after the last SPIxSCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPIx_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPIx_RDBR, because of the latency. Therefore, for SPIx_BAUD = 2 or SPIx_BAUD = 3, RXS must be set before SPIF to read SPIx_RDBR. For larger SPIx_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = b#00 mode may be the best operation option. In this mode, software performs a dummy read from the SPIx_RDBR register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPIx_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 SPIx_RDBR register does not initiate another transfer. It is recommended to disable the SPI port before the final SPIx_RDBR read access. Reading the SPIx_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 is 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 = b#00 or TIMOD = b#10 mode. With TIMOD = b#01 or TIMOD = b#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 PORTx_FER and/or PORTx_MUX registers to properly configure the required PBx, PEx, and/or PGx pins for SPI use.
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 7, "Direct Memory Access".
3. The processor core writes to the SPIx_FLG register, setting one or more of the SPI flag select bits (FLSx).
4. The processor core writes to the SPIx_BAUD and SPIx_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 = b#10) or "transmit with DMA" (TIMOD = b#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 SPIx_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 SPIx_TDBR register, it initiates a transfer on the SPI link.

6. The SPI then generates the programmed clock pulses on SPIxSCK and simultaneously shifts data out of SPIxMOSI and shifts data in from SPIxMISO. For receive transfers, the value in the shift register is loaded into the SPIx_RDBR register at the end of the transfer. For transmit transfers, the value in the SPIx_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 22-9 on page 22-40 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 SPIxMISO pin, overwriting the older data in the SPIx_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPIx_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ = 1, the device repeatedly transmits 0s on the SPIxMOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPIx_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, 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 SPIx_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 SPIx_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPIx_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPIx_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#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 $\overline{\text{SPIxSS}}$ signal to the active-low state or by the first active edge of SPIxSCK, 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 PORTx_FER and PORTx_MUX registers to properly configure the GPIO pins as SPI signals.
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 7, "Direct Memory Access".
3. The processor core writes to the SPIx_CTL register to define the mode of the serial link to be the same as the mode setup in the SPI master. The TIMOD field will be configured to select either "receive with DMA" (TIMOD = b#10) or "transmit with DMA" (TIMOD = b#11) mode.
4. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SPIxSCK edges. The value in the shift register is loaded into the SPIx_RDBR register at the end of the transfer. As the SPI reads data from the SPIx_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPIx_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on active SPIxSCK edges. The value in the SPIx_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 SPIxSCK 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 SPIxSCK edges as long as the slave select input is active.

See Figure 22-9 on page 22-40 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 SPIxMOSI pin, overwriting the older data in the SPIx_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPIx_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits 0s on the SPIxMISO pin. If SZ = 0, it repeatedly transmits the contents of the SPIx_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits 0s on the SPIxMISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored

when configured in transmit DMA mode, including the data in the SPIx_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPIx_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPIx_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPIx_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10), or when there is a TXE underflow error condition (when TIMOD = b#11).

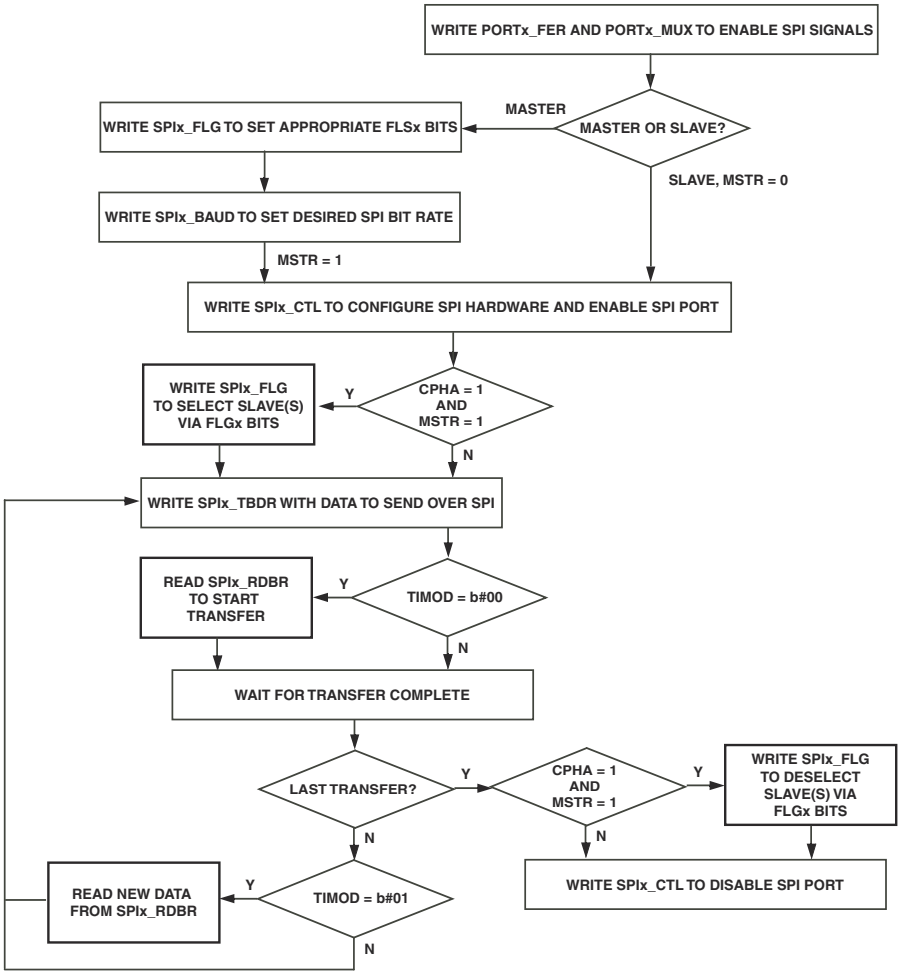


Figure 22-8. Core-Driven SPI Flow Chart

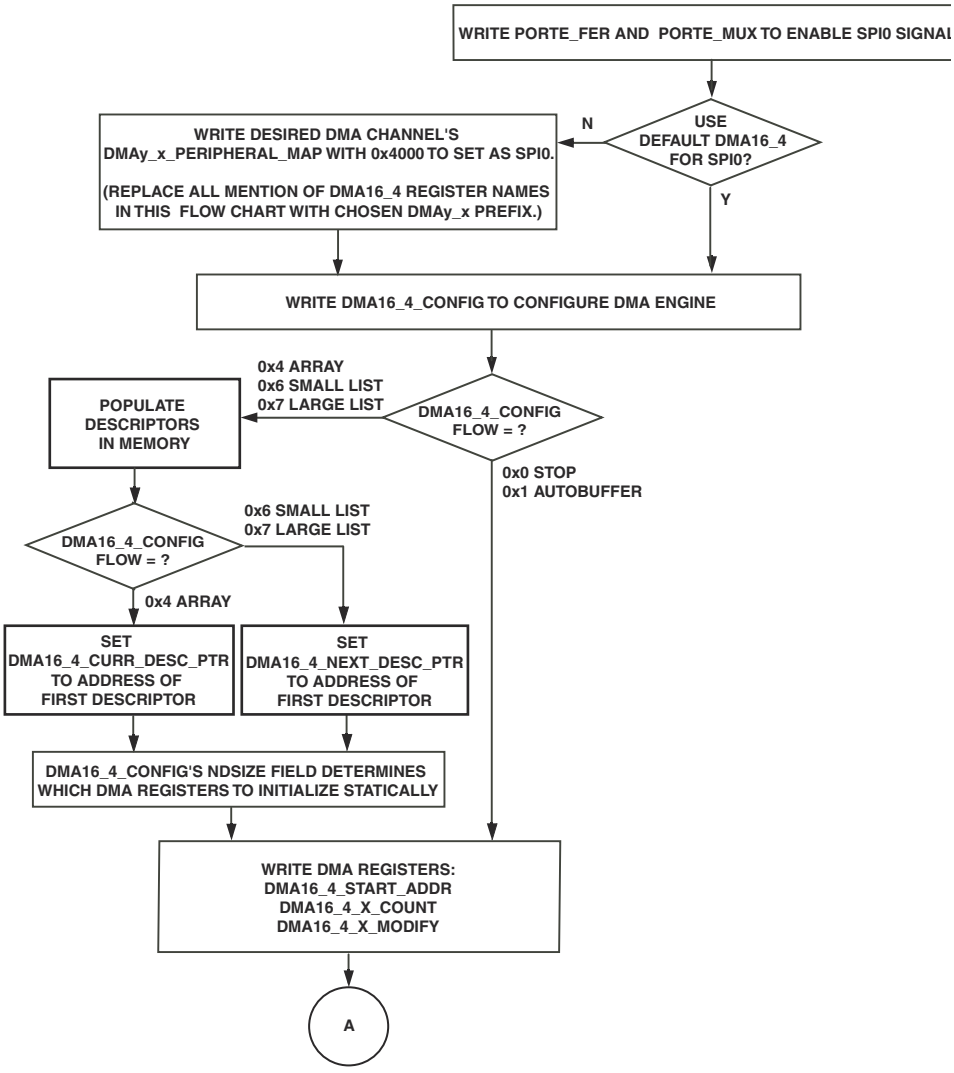


Figure 22-9. SPI0 DMA Flow Chart (Part 1 of 3)

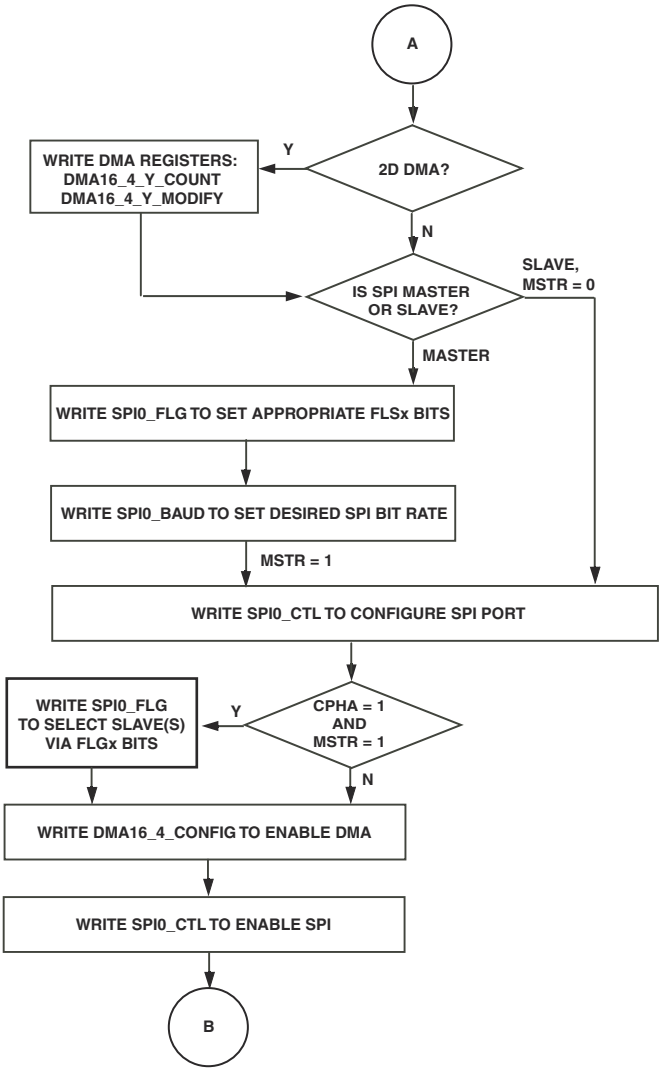


Figure 22-10. SPI0 DMA Flow Chart (Part 2 of 3)

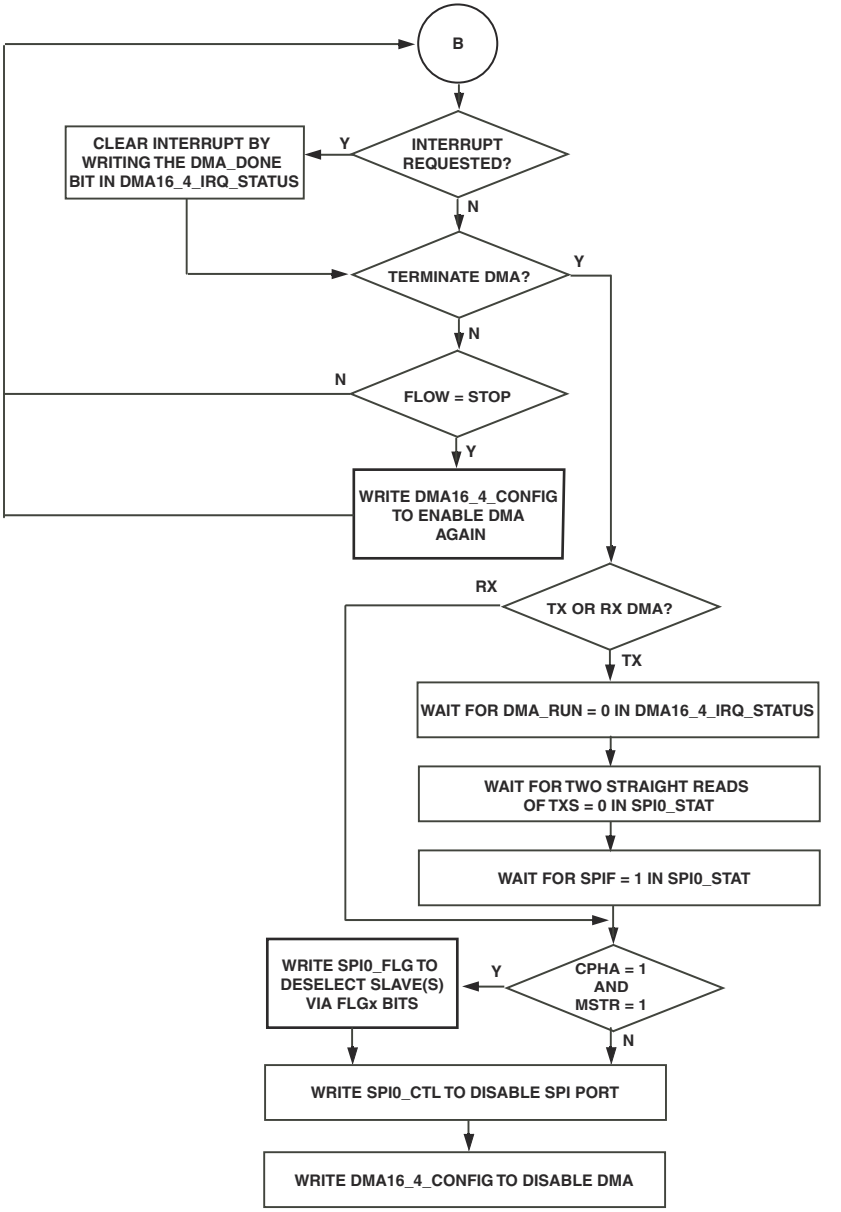


Figure 22-11. SPI0 DMA Flow Chart (Part 3 of 3)

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: `SPIx_BAUD`, `SPIx_CTL`, `SPIx_FLG`, and `SPIx_STAT`. Two registers are used for buffering receive and transmit data: `SPIx_RDBR` and `SPIx_TDBR`. For more information about DMA-related registers see Chapter 7, "Direct Memory Access".

See "Error Signals and Flags" on page 22-23 for more information about how the bits in these registers are used to signal errors and other conditions.

Table 22-8 shows the functions of the SPI registers. Figure 22-12 through Figure 22-18 on page 22-49 provide details.

Table 22-8. SPI Registers

Register Name	Description	Notes
SPIx_BAUD	SPIx port baud rate registers on page 22-44	Value of 0 or 1 disables the serial clock
SPIx_CTL	SPIx port control registers on page 22-45	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPIx_FLG	SPIx port flag registers on page 22-46	Bits 0 and 8 are reserved
SPIx_STAT	SPIx port status registers on page 22-48	SPIF bit can be set by clearing SPE in SPIx_CTL
SPIx_TDBR	SPIx port transmit data buffer registers on page 22-48	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPIx_CTL)
SPIx_RDBR	SPIx port receive data buffer registers on page 22-49	When register is read, hardware events can be triggered
SPIx_SHADOW	SPIx port RDBR shadow registers on page 22-49	Register has the same contents as SPIx_RDBR, but no action is taken when it is read

SPI Baud Rate (SPIx_BAUD) Register

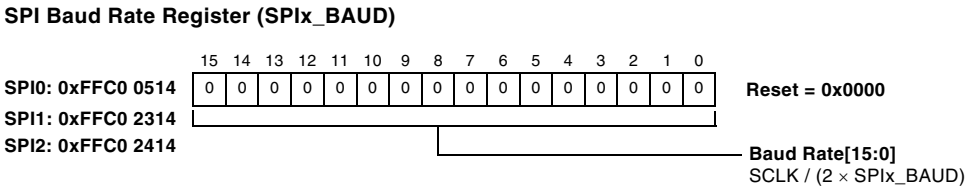


Figure 22-12. SPI Baud Rate Register

SPI Control (SPIx_CTL) Register

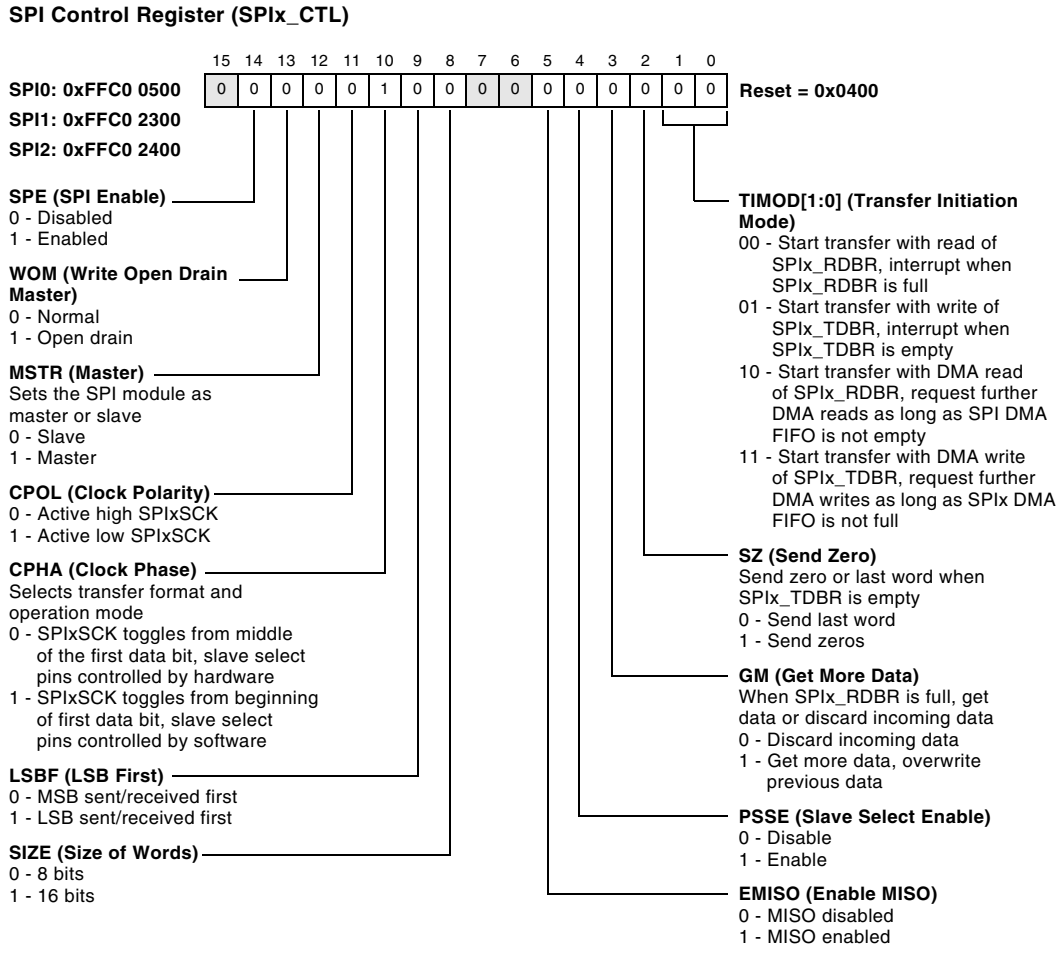


Figure 22-13. SPI Control Register

SPI Flag (SPIx_FLG) Register

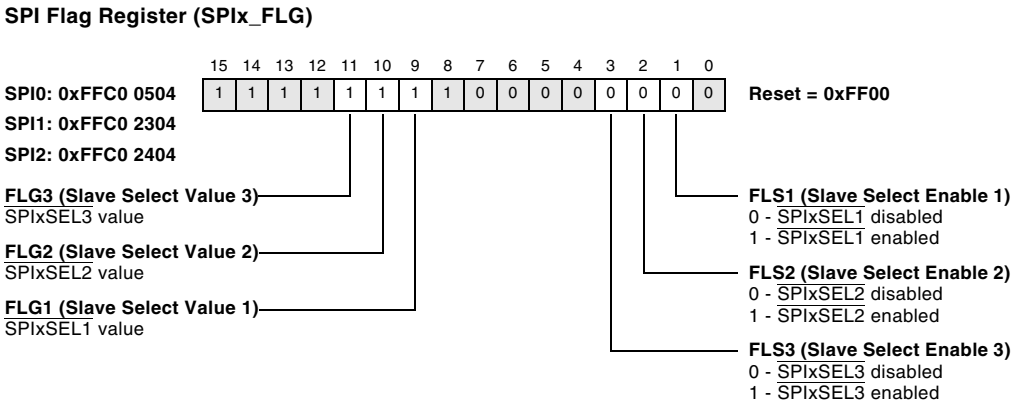


Figure 22-14. SPIx Flag Register

The SPIx_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 (PBx/PEx/PGx) pin. When an FLSx bit is set, the corresponding port pin is driven as a slave select. For example, if FLS1 is set in SPI0_FLG, PE4 is driven as a slave select ($\overline{\text{SPI0SEL1}}$). Table 22-4 on page 22-11 shows the association of the FLSx bits and the corresponding port pins.

  If the FLSx bit is not set, the general-purpose port registers (PORTxIO_DIR and others) configure and control the corresponding port pins.

- Slave select value (FLGx) bits

- When a PBx/PEx/PGx pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPIx_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 PE6 as a slave select, FLS3 in SPIO_FLG must be set. Clearing FLG3 in SPIO_FLG drives PE6 low; setting FLG3 drives PE6 high. The PE6 pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, PE6 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 PE6 as a slave select pin, it is only necessary to set the FLS3 bit in SPIO_FLG. It is not necessary to write to the FLG3 bit, because the SPI hardware automatically drives the PE6 pin.

SPI Status (SPIx_STAT) Register

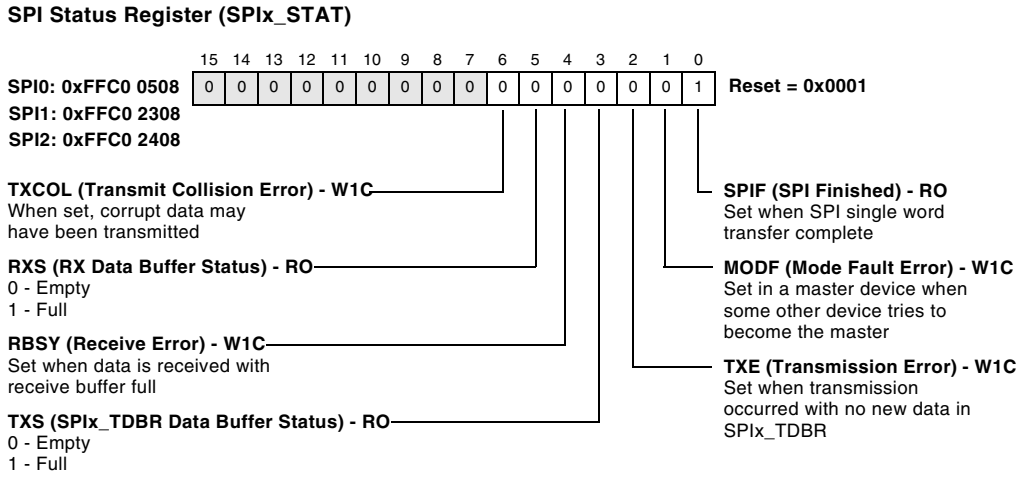


Figure 22-15. SPI Status Register

SPI Transmit Data Buffer (SPIx_TDBR) Register

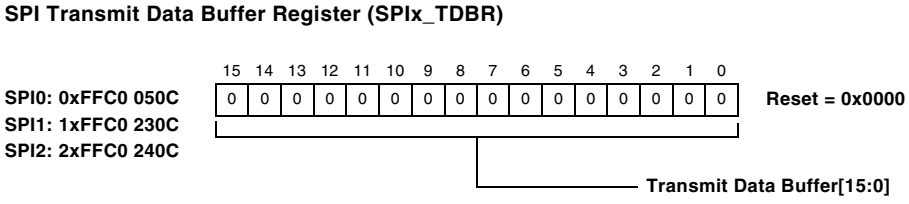


Figure 22-16. SPI Transmit Data Buffer Register

SPI Receive Data Buffer (SPIx_RDBR) Register

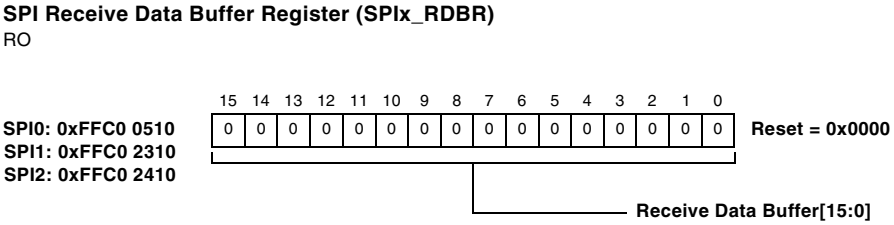


Figure 22-17. SPI Receive Data Buffer Register

SPI RDBR Shadow (SPIx_SHADOW) Register

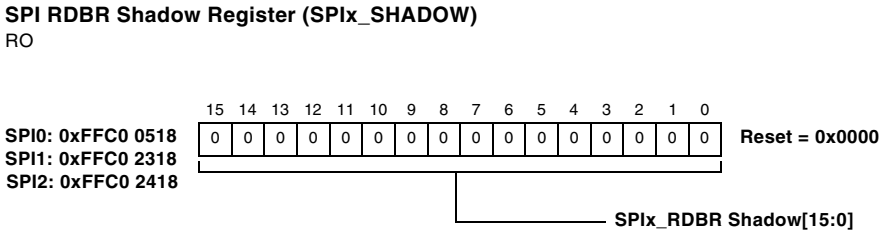


Figure 22-18. SPI RDBR Shadow Register

Programming Examples

This section includes examples (Listing 22-1 on page 22-50 through Listing 22-8 on page 22-57) for core generated transfer and for use with DMA. Each code example assumes that the appropriate `defBF54x` header file is included and that core writes to `PORTx_FER` and `PORTx_MUX` have been made to configure port pins associated with the SPI.

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 22-1. SPI Register Initialization

```
SPI0_Register_Initialization:
  P0.H = hi(SPI0_FLG);
  P0.L = lo(SPI0_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x3); /* FLS3 */
  W[P0] = R0; /* Enable slave-select output pin */

  P0.H = hi(SPI0_BAUD);
  P0.L = lo(SPI0_BAUD);
  R0.L = 0x208E;  /* Write to SPI Baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133 MHz, SPI clock ~= 8 kHz
*/

/* Setup SPI0 Control Register */
/***********************************************
 * TIMOD [1:0] = 00 : Transfer On RDBR Read.
 * SZ [2]      =  0 : Send Last Word When TDBR Is Empty
 * GM [3]      =  1 : Overwrite Previous Data If RDBR 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 : Hardware Controls Slave-Select Outputs
* CPOL [11]   =  1 : Active Low 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(SPI0_CTL) ;
 P0.L = lo(SPI0_CTL) ;
 R0 = 0x5908;
 W[P0] = R0.L; ssync; /* Enable SPI0 as MASTER */
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following a dummy read of `SPI0_RDBR`. Typically, known data which is desired to be transmitted to the slave is preloaded into the `SPI0_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 22-2. Initiate Transfer

```
Initiate_Transfer:
  P0.H = hi(SPI0_FLG);
  P0.L = lo(SPI0_FLG);
  R0 = W[P0] (Z);
  BITCLR (R0,0xB);  /* FLG3 */
  W[P0] = R0;  /* Drive 0 on enabled slave-select pin */
```

```
  P0.H = hi(SPI0_TDBR);  /* SPI Transmit Register */
  P0.L = lo(SPI0_TDBR);
  R0 = W[P1++] (z);  /* Get First Data To Be Transmitted And
Increment Pointer */
 W[P0] = R0;  /* Write to SPI0_TDBR */

  P0.H = hi(SPI0_RDBR);
  P0.L = lo(SPI0_RDBR);
  R0 = W[P0] (z);  /* Dummy read of SPI0_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 `SPI0_RDBR` initiates the next transfer.

Listing 22-3. SPI0 Interrupt Handler

```
SPI0_Interrupt_Handler:
Process_SPI0_Sample:
  P0.H = hi(SPI0_TDBR);  /* SPI0 transmit register */
  P0.L = lo(SPI0_TDBR);
  R0 = W[P1++](z);  /* Get next data to be transmitted */
  W[P0] = R0.l;  /* Write that data to SPI0_TDBR */

Kick_Off_Next:
  P0.H = hi(SPI0_RDBR);  /* SPI0 receive register */
  P0.L = lo(SPI0_RDBR);
  R0 = W[P0] (z);  /* Read SPI0 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 is sent in its entirety.

Listing 22-4. Stopping SPI

```
Stopping_SPI0:
  P0.H = hi(SPI0_CTL);
 P0.L = lo(SPI0_CTL);
  R0 = W[P0];
 BITCLR(R0, 14);  /* Clear SPI0 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 is 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 22-5. DMA Initialization

```
Initialize_DMA: /* DMA16_4 = default channel for SPI0 DMA */
  P0.H = hi(DMA16_4_CONFIG);
  P0.L = lo(DMA16_4_CONFIG);
  R0 = 0x1084(z); /* Autobuffer mode, IRQ on complete, linear
16-bit, mem read */
  w[P0] = R0;

  P0.H = hi(DMA16_4_START_ADDR);
  P0.L = lo(DMA16_4_START_ADDR);
  [p0] = p1;  /* Start address of TX buffer */

  P0.H  = hi(DMA16_4_X_COUNT);
  P0.L  = lo(DMA16_4_X_COUNT);
  R0    = NUM_SAMPLES;
  w[p0] = R0;  /* Number of samples to transfer */

  R0 = 2;
  P0.H = hi(DMA16_4_X_MODIFY);
  P0.L = lo(DMA16_4_X_MODIFY);
  w[p0] = R0;  /* 2 byte stride for 16-bit words */

  R0 = 1;  /* single dimension DMA means 1 row */
  P0.H = hi(DMA16_4_Y_COUNT);
  P0.L = lo(DMA16_4_Y_COUNT);
  w[p0] = R0;
```

SPI Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 22-6. SPI Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI0_FLG);
  P0.L = lo(SPI0_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x3); /* FLS3 */
  W[P0] = R0;  /* Enable slave-select output pin */

  P1.H = hi(SPI0_BAUD);
  P1.L = lo(SPI0_BAUD);
  R0.L   = 0x208E;  /* Write to SPI0 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 TDBR is empty
 * GM [3]      =  1 : Discard incoming data if RDBR is full
 * PSSE [4]    =  0 : Disables slave-select as input (master)
 * EMISO [5]   =  0 : SPIxMISO 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 : Hardware Controls Slave-Select Outputs
 * CPOL [11]   =  1 : Active LOW serial clock
 * MSTR [12]   =  1 : Device is master
 * WOM [13]    =  0 : Normal MOSI/MISO data output
 *                    (no open drain)
 * SPE [14]    =  0 : SPI module is disabled
 * [15]        =  0 : RESERVED
 ************************************************/
  /* Configure SPI0 as MASTER */
  R1 = 0x190B(z);  /* Leave disabled until DMA is enabled*/
```

```
  P1.L = lo(SPI0_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 22-7. Starting a Transfer

```
Initiate_Transfer:
  P0.H = hi(DMA16_4_CONFIG);
  P0.L = lo(DMA16_4_CONFIG);
  R2 = w[P0](z);
  BITSET (R2, 0);  /*Set DMA enable bit */
  w[p0] = R2.L;  /* Enable TX DMA */

  P4.H = hi(SPI0_CTL);
  P4.L = lo(SPI0_CTL);
  R2=w[p4](z);
  BITSET (R2, 14);  /* Set SPI0 enable bit */
  w[p4] = R2;  /* Enable SPI0 */
```

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, `SPI0_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 is shifted out. At that point, it is safe to shut down the SPI port and the DMA engine.

Listing 22-8. Stopping a Transfer

```
SPI_DMA_INTERRUPT_HANDLER:
  P0.L = lo(DMA16_4_IRQ_STATUS);
  P0.H = hi(DMA16_4_IRQ_STATUS);
  R0 = 1 ;
  W[P0] = R0 ;  /* Clear DMA interrupt */

  /* Wait for DMA to complete */
  P0.L = lo(DMA16_4_IRQ_STATUS);
  P0.H = hi(DMA16_4_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(SPI0_STAT);
  P0.H = hi(SPI0_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(SPI0_CTL);
  P0.H = hi(SPI0_CTL);
  R0 = W[P0] (Z);
  BITCLR (R0,0xe); /* Clear SPI enable bit */
  W[P0] = R0;  /* Disable SPI */

Disable_DMA:
  P0.L = lo(DMA16_4_CONFIG);
  P0.H = hi(DMA16_4_CONFIG);
  R0 = W[P0](Z);
  BITCLR (R0,0x0);  /* Clear DMA enable bit */
  W[P0] = R0;  /* Disable DMA */

  RTI;  /* Exit Handler */
```

23 TWO-WIRE INTERFACE CONTROLLERS

ADSP-BF54x processor processors include two 2-wire interface (TWI) controllers. These controllers allow 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 chapter contains the following sections:

- "Overview" on page 23-2
- "Interface Overview" on page 23-3
- "Description of Operation" on page 23-6
- "TWI General Operation" on page 23-11
- "Functional Description" on page 23-13
- "Programming Model" on page 23-23
- "TWI Registers" on page 23-25
- "Programming Examples" on page 23-52
- "Electrical Specifications" on page 23-63

Overview

Each 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.

Each TWI externally moves 8-bit data while maintaining compliance with the I^2C bus protocol. The TWI controllers include these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master bus arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up
- Input filter for spike suppression
- Serial camera control bus support as specified in *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*.

Interface Overview

Figure 23-1 provides a block diagram of the TWI controllers. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCLx rate, to and from other TWI devices. The SCLx synchronizes the shifting and sampling of the data on the serial data pin.

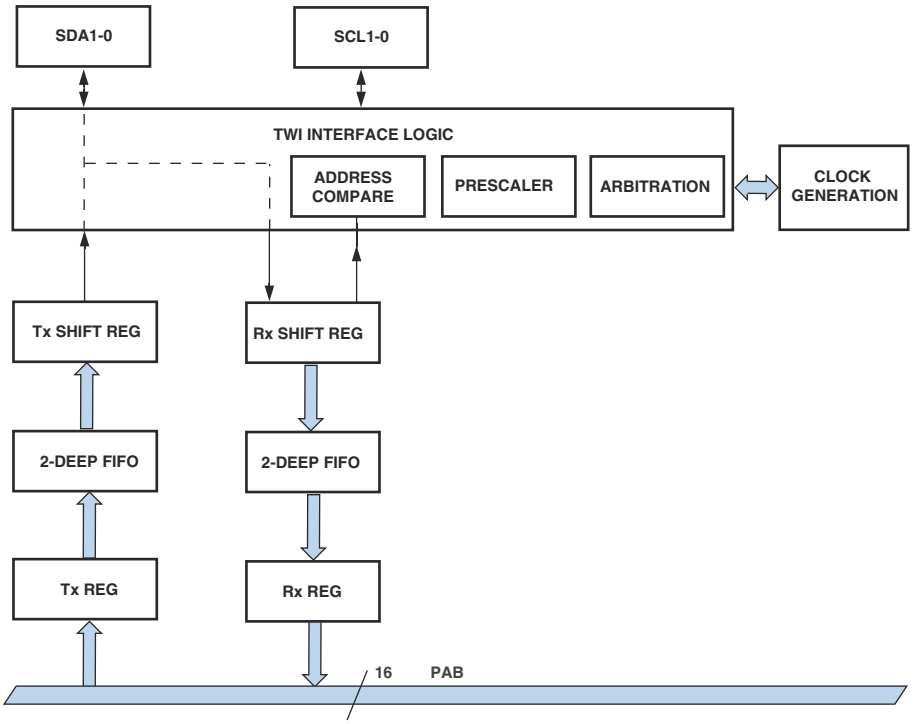


Figure 23-1. TWI Block Diagram

External Interface

The TWI signals are functionally multiplexed as general-purpose I/O. These signals, SDAx (serial data) and SCLx (serial clock) are open drain and as such require pull up resistors.

Serial Clock signal (SCL1–0)

In slave mode this signal is an input and an external master is responsible for providing the clock.

In master mode the TWI controllers must set this signal to the desired frequency. The TWI controllers support the standard mode of operation (up to 100 KHz) or fast mode (up to 400 KHz).

The TWI control register (TWIx_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.

$\text{PRESCALE} = f_{SCLK}/10\text{MHz}$

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.

It is not always possible to achieve 10 MHz accuracy. In such cases, it is safe to round up the PRESCALE value to the next highest integer. For example, if SCLK is 133 MHz, the PRESCALE value is calculated as 133 MHz/10 MHz = 13.3. In this case, a PRESCALE value of 14 ensures that all timing requirements are met.

Serial data signal (SDA1–0)

This is a bidirectional signal on which serial data is transmitted or received depending on the direction of the transfer.

TWI Pins

Table 23-1 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 23-1. TWI Pins

Pin	Description
SDA1–0	In/Out TWI serial data, high impedance reset value.
SCL1–0	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 any of 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 SCLx clock generation.

The clock generation module is used to generate an external SCLx 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 controllers follow the transfer protocol of the *Philips I^2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 23-2.

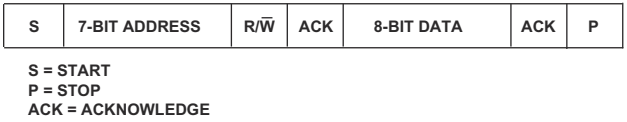


Figure 23-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, Figure 23-3 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 as a master. The slave has acknowledged both address and data.

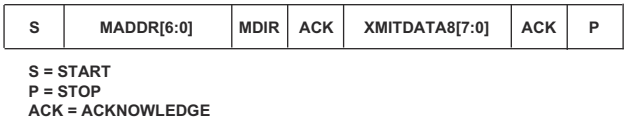


Figure 23-3. Data Transfer With Bit Illustration

Clock Generation and Synchronization

Each TWI controller implementation only issues a clock during master mode operation and only at the time a transfer is 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 23-4 for TWI controller 0.

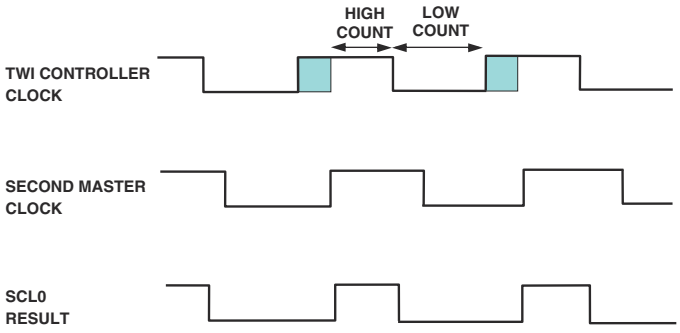


Figure 23-4. TWI Clock Synchronization

The TWI controller's serial clock (SCLx) 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 SCLx line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controllers initiate a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 23-5.

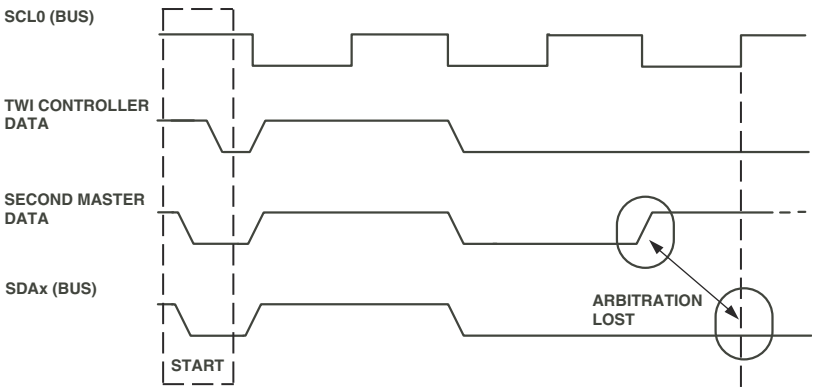


Figure 23-5. TWI Bus Arbitration

The TWI controller monitors the serial data bus (SDAx) while SCLx is high and if SDAx 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 SCLx is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controllers generate and recognize these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start "combined" transfers, as shown in Figure 23-6.

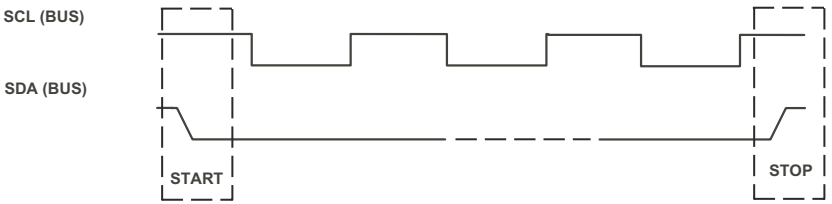


Figure 23-6. TWI Start and Stop Conditions

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 controllers always decode and acknowledge a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). general call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controllers are to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

The byte following the general call address usually defines what action needs to be taken by the slaves in response to the call. The command in the second byte is interpreted based on the value of its LSB. For a TWI slave device, this is not applicable, and the bytes received after the general call address are considered data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most effected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition (t_{BUF})

TWI General Operation

The following sections describe the general operation of the TWI controllers.

TWI Control

The TWI control register (TWIx_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 TWIx_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 controllers always generate an acknowledge in slave mode.

For either master and/or slave mode of operation, the TWI controllers are enabled by setting the TWIx_ENA bit in the TWIx_CONTROL register. It is recommended that this bit be set at the time PRESCALE is initialized and remains set. This guarantees accurate operation of bus busy detection logic.

The PRESCALE field of the TWIx_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 SCLx is an output in master mode and an input in slave mode.

During master mode operation, the SCLx clock divider register (TWIx_CLKDIV) values are used to create the high and low durations of the serial clock (SCLx). 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 SCLx period / 10 MHz time reference

For example, for an SCLx 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 SCLx 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 TWIx_CLKDIV register specifies the number of 10 MHz time reference periods the serial clock (SCLx) 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 TWIx_CLKDIV register number of internal time reference periods the serial clock (SCLx) is held low. It is represented as an 8-bit binary value.

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 TWIx_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 TWIx_ENA bit enable must be set. Note once the TWI controller is enabled a bus busy condition may be detected. This condition should clear after t_{BUF} has expired assuming no additional bus activity is 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 TWIx_SLAVE_ADDR. The appropriate 7 bits are used in determining a match during the address phase of the transfer.
2. Program TWIx_XMT_DATA8 or TWIx_XMT_DATA16. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (SCLx) is stretched and an interrupt is generated until data is written to the transmit FIFO.

3. Program TWIx_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 TWIx_SLAVE_CTL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 23-2 shows what the interaction between the TWI controllers and the processor might look like using this example.

Table 23-2. Slave Mode Setup Interaction

TWI Controller	Processor
Interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: RCVSERV – Receive buffer is full.	Acknowledge: Clear interrupt source bits. Read TWIx_FIFO_STAT. Read receive FIFO buffer.
...	...
Interrupt: SCOMP – Slave transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

- Program TWIx_CLKDIV. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program `TWIx_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWIx_XMT_DATA8` or `TWIx_XMT_DATA16`. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.
3. Program `TWIx_FIFO_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 `TWIx_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 `TWIx_MASTER_CTL`. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 23-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 23-3. Master Mode Transmit Setup Interaction

TWI Controller	Processor
Interrupt: XMTSERV – Transmit buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.

Table 23-3. Master Mode Transmit Setup Interaction (Cont'd)

TWI Controller	Processor
...	...
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 `TWIx_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWIx_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 `TWIx_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 `TWIx_MASTER_CTL`. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

After the `TWI_DCNT` bit is decremented to zero, the TWI master device sends a NAK to indicate to the slave transmitter that the bus should be released. This allows the master to send the STOP signal to terminate the transfer.

Table 23-4 shows what the interaction between the TWI controllers and the processor might look like using this example.

Table 23-4. Master Mode Receive Setup Interaction

TWI Controller	Processor
Interrupt: RCVSERV – Receive buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Clock Stretching

Clock stretching is an added functionality of the TWI controller in master mode operation. This new behavior utilizes self-induced stretching of I^2C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this. TWI Controller as master supports three modes of clock stretching:

- "Clock Stretching During FIFO Underflow" on page 23-18
- "Clock Stretching during FIFO Overflow" on page 23-19
- "Clock Stretching During Repeated Start Condition" on page 23-21

Clock Stretching During FIFO Underflow

During a master mode transmit an interrupt is generated at the instant the transmit FIFO becomes empty. At this time the most recent byte begins transmission. If the XMTSERV interrupt is not serviced, the concluding acknowledge phase of the transfer is stretched. Stretching of the clock continues until new data bytes are written to the transmit FIFO (`TWIx_XMT_DATA8` or `TWIx_XMT_DATA16`). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (`DCNT` = 0) at which time the transmission is concluded (`MCOMP`) as shown in Figure 23-7 and described in Table 23-5.

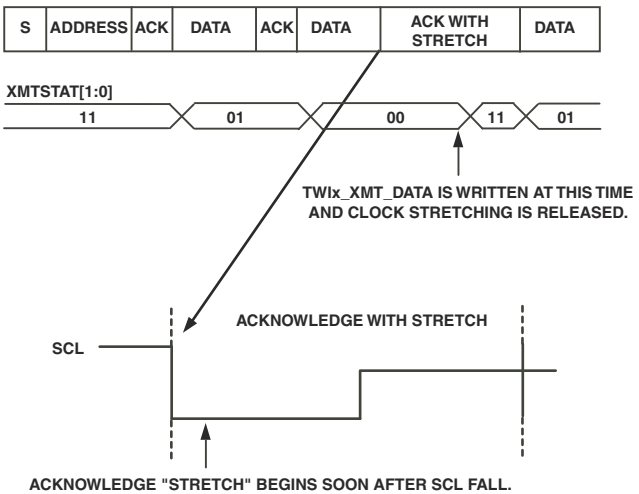


Figure 23-7. Clock Stretching during FIFO Underflow

Table 23-5. FIFO Underflow Case

TWI Controller	Processor
Interrupt: XMTSERV – Transmit FIFO buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).	Acknowledge: Clear interrupt source bits.

Clock Stretching during FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (`TWIx_RCV_DATA8`, `TWIx_RCV_DATA16`). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (`DCNT` = 0x00) at which time the reception is concluded (`MCOMP`) as shown in Figure 23-8 and described in Table 23-6.

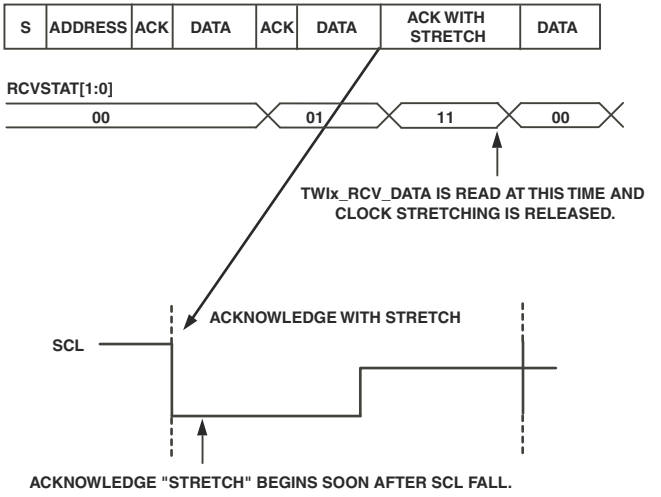


Figure 23-8. Clock Stretching During FIFO Overflow

Table 23-6. FIFO Overflow Case

TWI Controller	Processor
Interrupt: RCVSERV – Receive FIFO buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Clock Stretching During Repeated Start Condition

The repeated start feature in I^2C protocol requires transitions between two subsequent transfers. With the use of clock stretching the task of managing these transitions becomes simpler, and common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock initiates a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer generates a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (DCNT = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 23-9 and described in Table 23-7.

Table 23-7. Repeated Start Case

TWI Controller	Processor
Interrupt: MCOMP – Initial transmit has completed and DCNT = 0x00. Note: transfer in progress, RSTART previously set.	Acknowledge: Clear interrupt source bits. Write TWIx_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).
Interrupt: RCVSERV – Receive FIFO is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

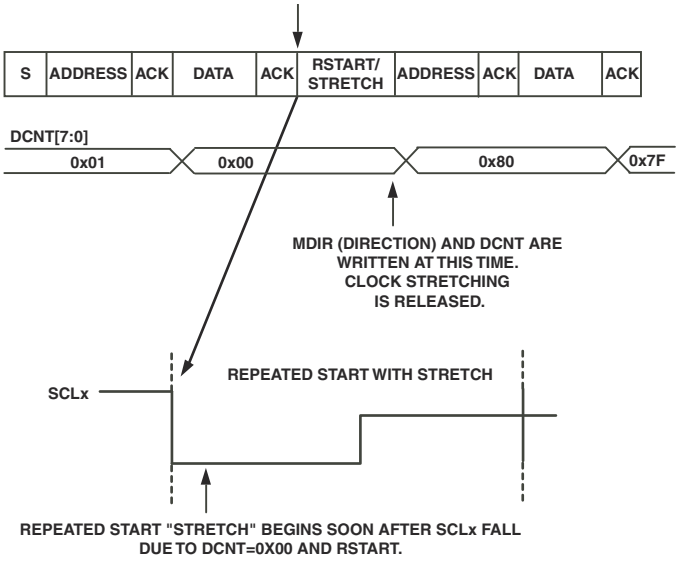


Figure 23-9. Clock Stretching during Repeated Start Condition

Programming Model

Figure 23-10 and Figure 23-11 illustrate the programming model for the TWI.

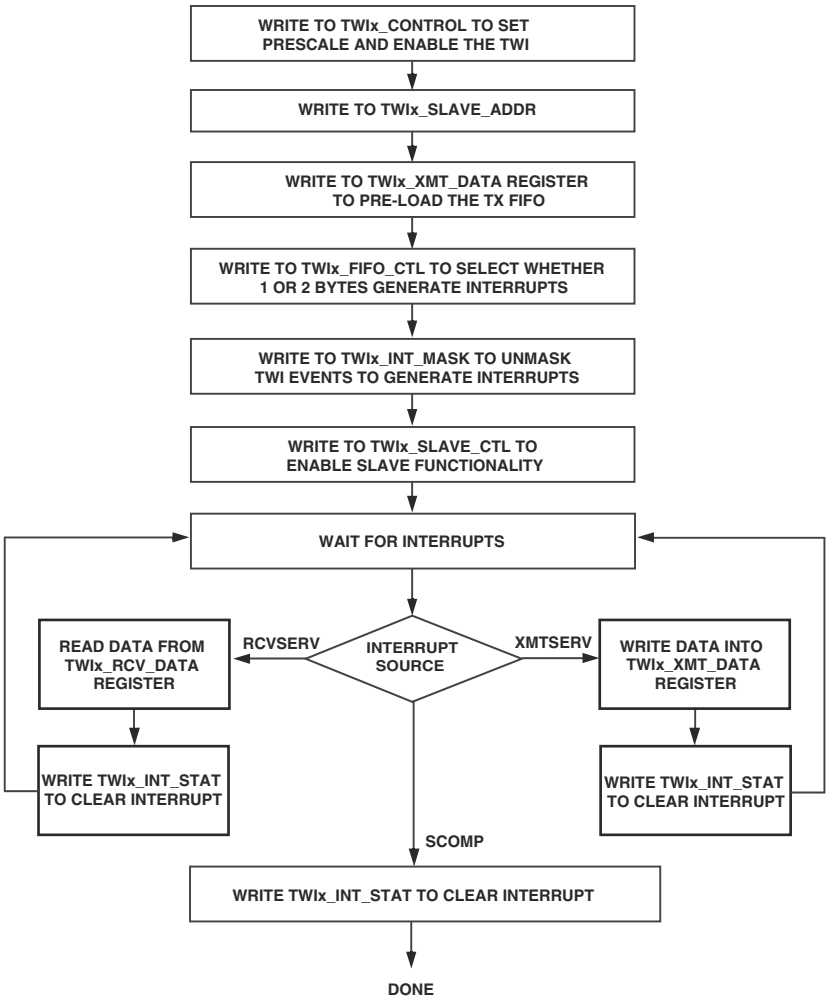


Figure 23-10. TWI Slave Mode

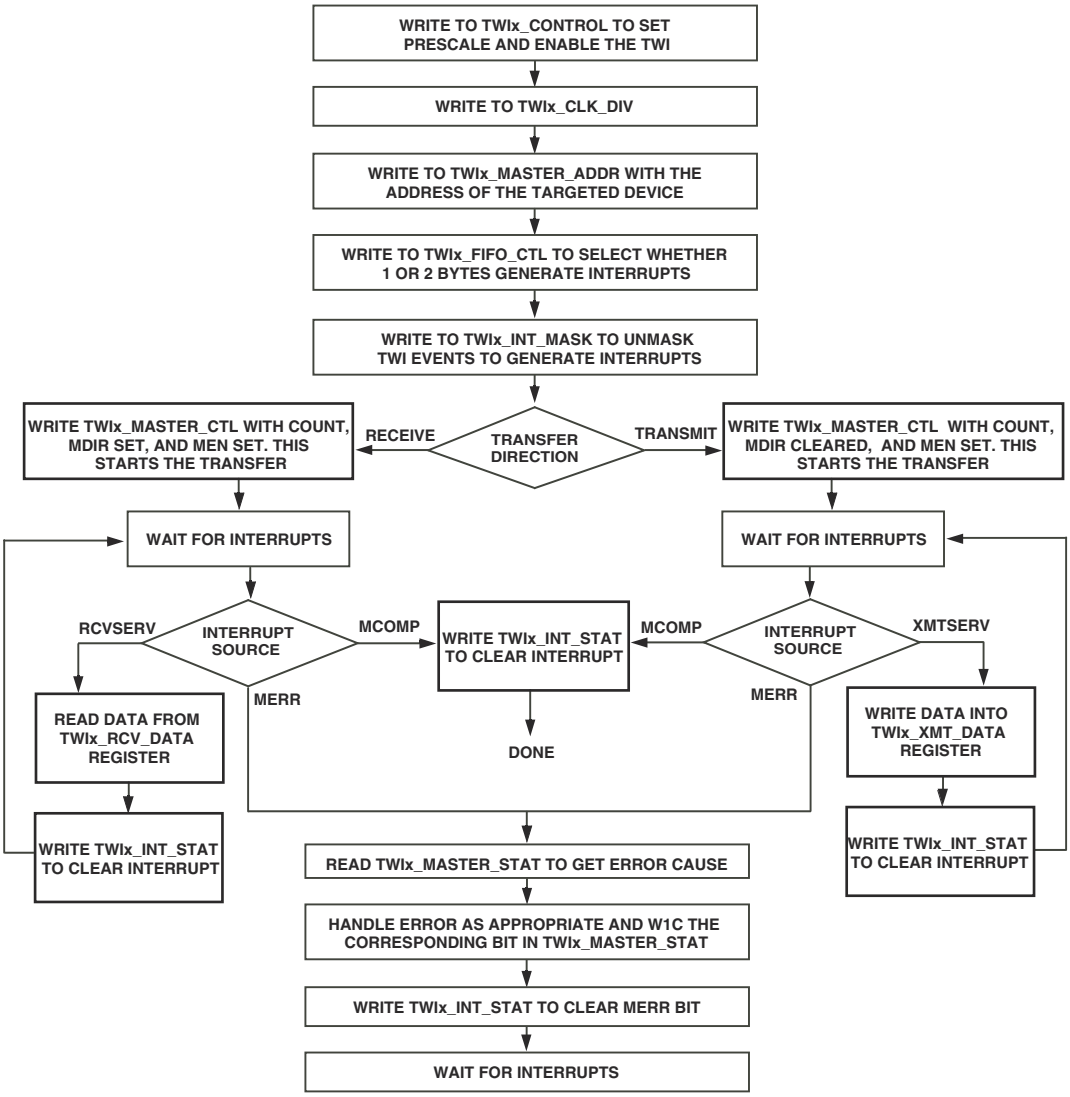


Figure 23-11. TWI Master Mode

TWI Registers

The TWI controller has 16 registers described in the following sections. Table 23-8 lists the TWI registers.

Table 23-8. TWIx Registers

TWI0 Memory Mapped Address	Register Name	Description
0xFFC0 0700	TWIx_CLKDIV	"SCLx Clock Divider (TWIx_CLKDIV) Register" on page 23-26
0xFFC0 0704	TWIx_CONTROL	"TWI Control (TWIx_CONTROL) Register" on page 23-27
0xFFC0 0708	TWIx_SLAVE_CTL	"TWI Slave Mode Control (TWIx_SLAVE_CTL) Register" on page 23-27
0xFFC0 0710	TWIx_SLAVE_ADDR	"TWI Slave Mode Address (TWIx_SLAVE_ADDR) Register" on page 23-30
0xFFC0 070C	TWIx_SLAVE_STAT	"TWI Slave Mode Status (TWIx_SLAVE_STAT) Register" on page 23-30
0xFFC0 0714	TWIx_MASTER_CTL	"TWI Master Mode Control (TWIx_MASTER_CTL) Register" on page 23-32
0xFFC0 071C	TWIx_MASTER_ADDR	"TWI Master Mode Address (TWIx_MASTER_ADDR) Register" on page 23-35
0xFFC0 0718	TWIx_MASTER_STAT	"TWI Master Mode Status (TWIx_MASTER_STAT) Register" on page 23-35
0xFFC0 0720	TWIx_INT_STAT	"TWI Interrupt Status (TWIx_INT_STAT) Register" on page 23-44
0xFFC0 0724	TWIx_INT_MASK	"TWI Interrupt Mask (TWIx_INT_MASK) Register" on page 23-43
0xFFC0 0728	TWIx_FIFO_CTL	"TWI FIFO Control (TWIx_FIFO_CTL) Register" on page 23-39
0xFFC0 072C	TWIx_FIFO_STAT	"TWI FIFO Status (TWIx_FIFO_STAT) Register" on page 23-41

Table 23-8. TWIx Registers (Cont'd)

TWI0 Memory Mapped Address	Register Name	Description
0xFFC0 0780	TWIx_XMT_DATA8	"TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Register" on page 23-48
0xFFC0 0784	TWIx_XMT_DATA16	"TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Register" on page 23-49
0xFFC0 0788	TWIx_RCV_DATA8	"TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Register" on page 23-50
0xFFC0 078C	TWIx_RCV_DATA16	"TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Register" on page 23-51

SCLx Clock Divider (TWIx_CLKDIV) Register

The TWIx_CLKDIV register values are used during master mode operation to create the high and low durations of the serial clock (SCLx). 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.

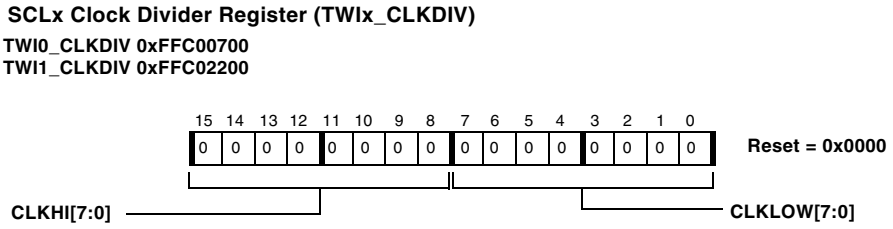


Figure 23-12. SCLx Clock Divider Register

TWI Control (TWIx_CONTROL) Register

The TWIx_CONTROL register is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

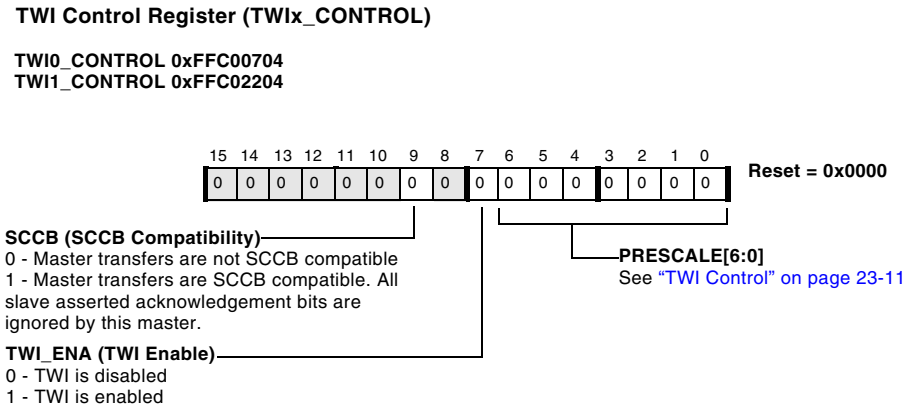


Figure 23-13. TWI Control Register

TWI Slave Mode Control (TWIx_SLAVE_CTL) Register

The TWIx_SLAVE_CTL register 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.

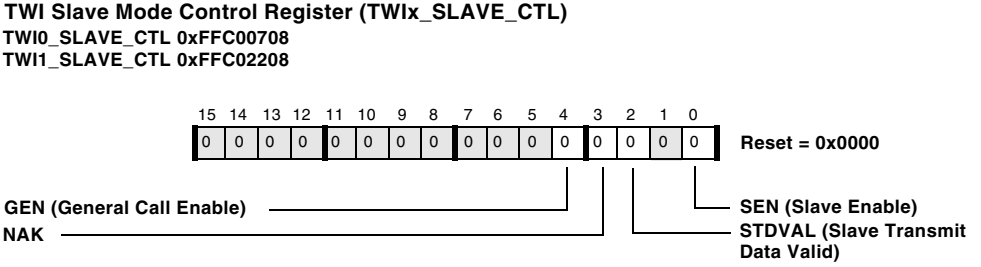


Figure 23-14. TWI Slave Mode Control Register

Additional information for the TWIx_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. At the time the NAK bit was set, a data byte may have been in the process of being received. The byte is NAK'd (not acknowledged) back to the master yet the byte is accepted into the receive FIFO and receive FIFO status (RCVSTAT) is updated accordingly.

[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 Mode Address (TWIx_SLAVE_ADDR) Register

The TWIx_SLAVE_ADDR register 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.

TWI Slave Mode Address Register (TWIx_SLAVE_ADDR)

TWI0_SLAVE_ADDR 0xFFC00710
TWI1_SLAVE_ADDR 0xFFC02210

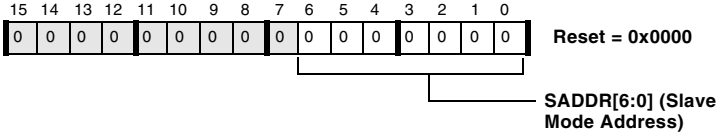


Figure 23-15. TWI Slave Mode Address Register

TWI Slave Mode Status (TWIx_SLAVE_STAT) Register

During and at the conclusion of slave mode transfers, the TWIx_SLAVE_STAT register 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.

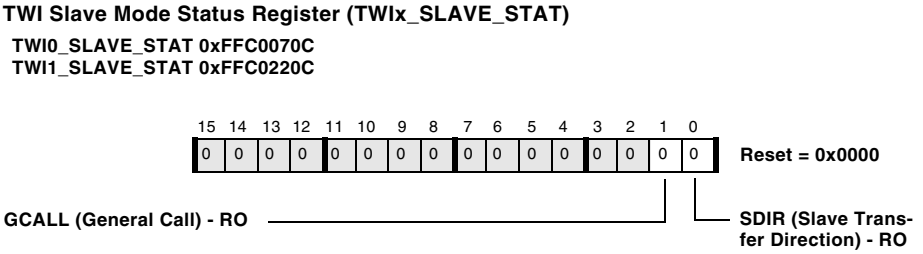


Figure 23-16. TWI Slave Mode Status Register

- **General call** (GCALL)

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the address was determined to be a general call.

  [0] At the time of addressing, the address was not determined to be a general call.

- **Slave transfer direction** (SDIR)

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

TWI Master Mode Control (TWIx_MASTER_CTL) Register

The TWIx_MASTER_CTL register 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.

TWI Master Mode Control Register (TWIx_MASTER_CTL)
TWI0_MASTER_CTL 0xFFC00714
TWI1_MASTER_CTL 0xFFC02214

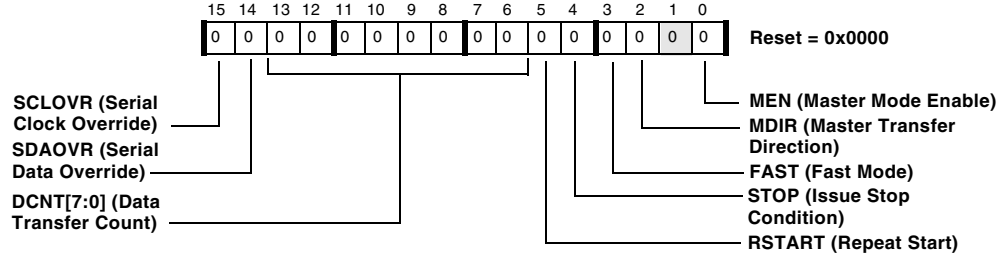


Figure 23-17. TWI Master Mode Control Register

Additional information for the TWIx_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. In the event a master transmit is aborted due to a slave data NAK, the value of DCNT equals the number of bytes not sent. The byte which was NAK'd by the slave is counted as a byte which was sent.

- **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 (TWIx_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 (after the DCNT bit decrements to zero), including transfers terminated due to errors.

  [1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDAx, SCLx) are no longer driven. Write-1-to-clear status bits are not affected.

TWI Master Mode Address (TWIx_MASTER_ADDR) Register

During the transmit phase of a transfer, the TWI controllers, with their master enabled, transmits the contents of the TWIx_MASTER_ADDR register. 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 TWIx_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the master mode control register.

TWI Master Mode Address Register (TWIx_MASTER_ADDR)
TWI0_MASTER_ADDR 0xFFC0071C
TWI1_MASTER_ADDR 0xFFC0221C

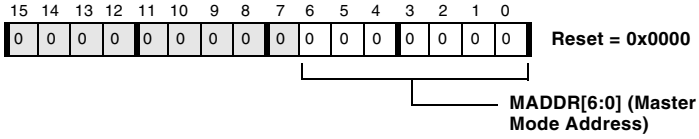


Figure 23-18. TWI Master Mode Address Register

TWI Master Mode Status (TWIx_MASTER_STAT) Register

The TWIx_MASTER_STAT register 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.

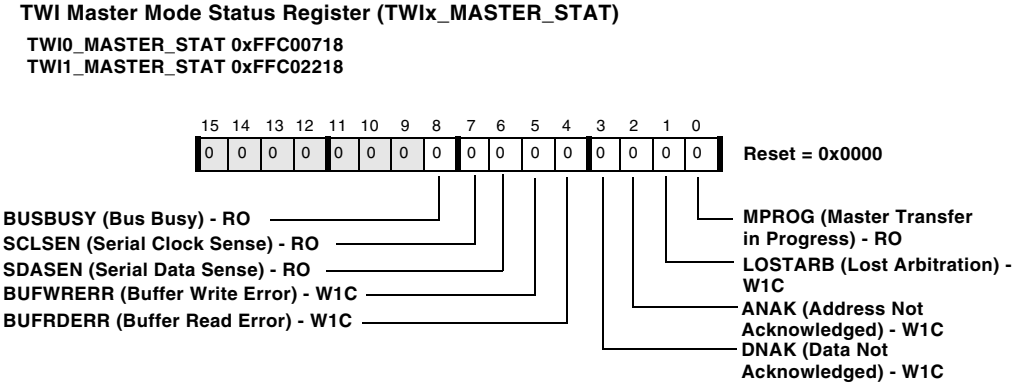


Figure 23-19. TWI Master Mode Status Register

- **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 is 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. Buffer write error, under normal operation, is never set due to the master’s ability to stretch the clock and avoid the circumstances described here. 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. Buffer read error, under normal operation, is never set due to the master's ability to stretch the clock and avoid the circumstances described here. 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 transfer 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 transfer 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 FIFO Control (TWIx_FIFO_CTL) Register

The TWIx_FIFO_CTL register control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

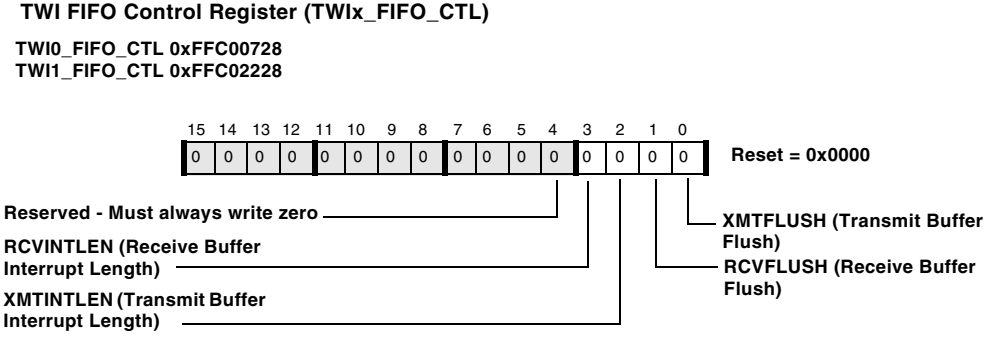


Figure 23-20. TWI FIFO Control Register

Additional information for the TWIx_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 TWIx_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 (b#01 or b#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 TWIx_FIFO_STAT register indicates two bytes in the FIFO are empty (b#00).

  [0] An interrupt (XMTSERV) is set when XMTSTAT indicates one or two bytes in the FIFO are empty (b#01 or b#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 Status (TWIx_FIFO_STAT) Register

TWI FIFO Status Register (TWIx_FIFO_STAT)
All bits are RO.

TWI0_FIFO_STAT 0xFFC0072C
TWI1_FIFO_STAT 0xFFC0222C

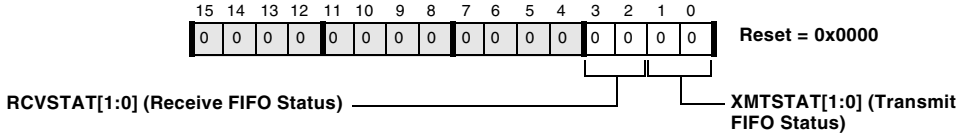


Figure 23-21. TWI FIFO Status Register

TWI FIFO Status

The fields in the `TWIx_FIFO_STAT` register 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.

  [b#11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.

  [b#10] Reserved

  [b#01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

  [b#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.

  [b#11] The FIFO is full and contains two bytes of data.

  [b#10] Reserved

  [b#01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [b#00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

TWI Interrupt Mask (TWIx_INT_MASK) Register

The TWIx_INT_MASK register enables interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the TWI interrupt status (TWIx_INT_STAT) register. Reading and writing the TWI interrupt mask register does not affect the contents of the TWI interrupt status register.

TWI Interrupt Mask Register (TWIx_INT_MASK)
For all bits, 0 = Interrupt generation disabled, 1 = Interrupt generation enabled.

TWI0_INT_MASK 0xFFC00724
TWI1_INT_MASK 0xFFC02224

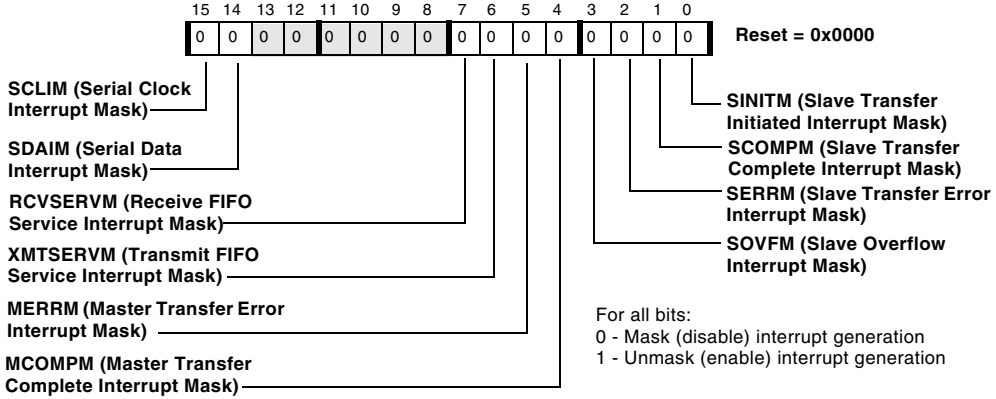


Figure 23-22. TWI Interrupt Mask Register

TWI Interrupt Status (TWIx_INT_STAT) Register

The TWIx_INT_STAT register 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.

TWI Interrupt Status Register (TWIx_INT_STAT)
All bits are sticky and W1C.

TWI0_INT_STAT 0xFFC00720
TWI1_INT_STAT 0xFFC02220

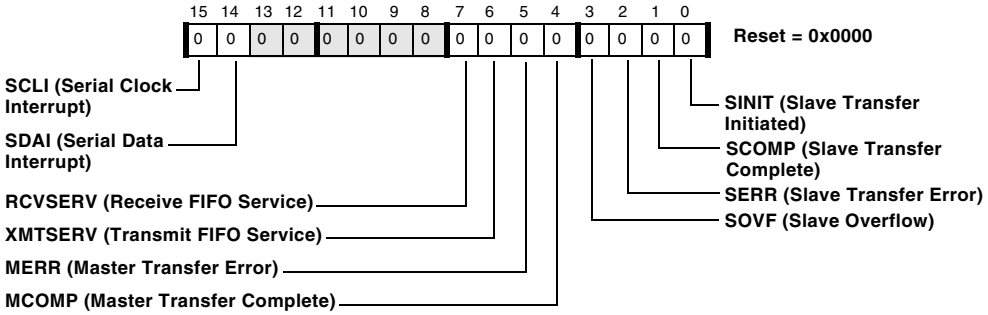


Figure 23-23. TWI Interrupt Status Register

- **Serial Clock Interrupt** (SCLI)

  If the TWI module is enabled (TWI_ENA), SCLI is set on a high-to-low transition of the serial clock pin (SCLx). Normally, this bit is not required for I^2C bus transfers. It is initially set on an I^2C transfer and does not require clearing.

  [1] A high-to-low transition was detected on the SCLx pin. This bit is W1C.

  [0] No transition was detected on the SCLx pin.

- **Serial Data Interrupt** (SDAI)

  If the TWI module is enabled (TWI_ENA), SDAI is set on a high-to-low transition of the serial data pin (SDAx). Normally, this bit is not required for I^2C bus transfers. It is initially set on an I^2C transfer and does not require clearing.

  [1] A high-to-low transition was detected on the SDAx pin. This bit is W1C.

  [0] No transition was detected on the SDAx pin.

- **Receive FIFO service** (RCVSERV)

  If RCVINTLEN in the TWIx_FIFO_CTL register is 0, this bit is set each time the RCVSTAT field in the TWIx_FIFO_STAT register is increased to either b#01 or b#11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to b#11.

  [0] The receive FIFO does not require servicing or the RCVSTAT field has not changed since this bit was last cleared.

  [1] The receive FIFO has one or two 8-bit locations available to be read.

- **Transmit FIFO service** (XMTSERV)

  If XMTINTLEN in the TWIx_FIFO_CTL register is 0, this bit is set each time the XMTSTAT field in the TWIx_FIFO_STAT register is updated to either b#01 or b#00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to b#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 (TWIx_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 is 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 is detected.

- **Slave transfer error** (SERR)

  [1] A slave error has occurred. A restart or stop condition has occurred during the data transmit 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 is initiated.

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Register

The TWIx_XMT_DATA8 register 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 TWIx_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWIx_FIFO_STAT 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 (TWIx_XMT_DATA8)

All bits are WO. This register always reads as 0x0000.

TWI0_XMT_DATA8 0xFFC00780
TWI1_XMT_DATA8 0xFFC02280

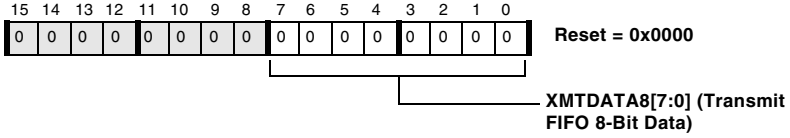


Figure 23-24. TWI FIFO Transmit Data Single Byte Register

TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Register

The TWIx_XMT_DATA16 register holds a 16-bit data value written into the FIFO buffer.

TWI FIFO Transmit Data Double Byte Register (TWIx_XMT_DATA16)
All bits are WO. This register always reads as 0x0000.

TWI0_XMT_DATA16 0xFFC00780
TWI1_XMT_DATA16 0xFFC02280

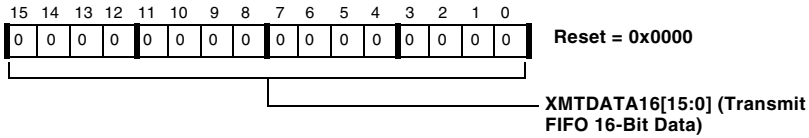


Figure 23-25. TWI FIFO Transmit Data Double Byte Register

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 23-26 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWIx_FIFO_STAT 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.

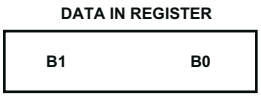


Figure 23-26. Little Endian Byte Order

TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Register

The TWIx_RCV_DATA8 register 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 TWIx_RCV_DATA8 accesses only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWIx_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 (TWIx_RCV_DATA8)
All bits are RO.

TWI0_RCV_DATA8 0xFFC00788
TWI1_RCV_DATA8 0xFFC02288

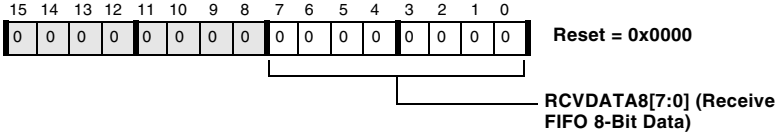


Figure 23-27. TWI FIFO Receive Data Single Byte Register

TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Register

The TWIx_RCV_DATA16 register holds a 16-bit data value read from the FIFO buffer.

TWI FIFO Receive Data Double Byte Register (TWIx_RCV_DATA16)
All bits are WO.

TWI0_RCV_DATA16 0xFFC0078C
TWI1_RCV_DATA16 0xFFC0228C

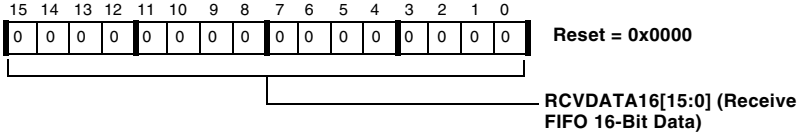


Figure 23-28. TWI FIFO Receive Data Double Byte Register

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 23-29 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWIx_FIFO_STAT 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.

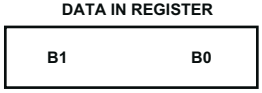


Figure 23-29. Little Endian Byte Order

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 23-1 shows how to initiate polled receive and transmit transfers in master mode.

Listing 23-1. Master Mode Receive/Transmit Transfer

```
/*********************************************************
 Macro for the Count field of the TWIx_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
*********************************************************/
```

```
TWIO_INIT:
/*********************************************************
Enable the TWIO 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 = TWIO_ENA | 0xA (z);
W[P1 + LO(TWIO_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(TWIO_CLKDIV)] = R5;

/*********************************************************
enable these signals to generate a TWIO interrupt: optional
*********************************************************/
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWIO_INT_MASK)] = R1;

/*********************************************************
The address needs to be shifted one place to the right,
for example, 1010 001x becomes 0101 0001 (0x51) the TWIO
controller actually send out 1010 001x where x is either a 0 for
writes or 1 for reads
```

```
************************************************************/
R6 = 0xBF;
R6 = R6 >> 1;
TWI0_INIT.END: W[P1 + LO(TWI0_MASTER_ADDR)] = R6;

/******************* END OF TWI0 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 kicks off the master transfer
************************************************************/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWI0_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(TWI0_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF !cc jump Rx_stat;
R0 = W[P1 + LO(TWI0_RCV_DATA16)](Z); /* Read data from the RX
fifo */
```

```
ssync;

/*************************************************************
check that master transfer has completed
MCOMP is set when Count reaches zero
*************************************************************/
M_COMP:
  R1 = W[P1 + LO(TWIO_INT_STAT)](z);
  CC = BITTST (R1, bitpos(MCOMP));
  if !CC jump M_COMP;
M_COMP.END:  W[P1 + LO(TWIO_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(TWIO_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 kicks off the transfer
```

```
*************************************************************/
R1 = TWICount(0xFE) | FAST | MEN;
W[P1 + LO(TWIO_MASTER_CTL)] = R1;
SSYNC;

/*************************************************************
  loop to write data to a TWIO slave device P3 times
*************************************************************/
P3 = length(TX_file);

LSETUP (Loop_Start1, Loop_End1) LC0 = P3;
  Loop_Start1:
    /*********************************************************
    check that there's at least one byte location empty in
    the tx fifo
    *********************************************************/
    XMTSERV_Status:
    R1 = W[P1 + LO(TWIO_INT_STAT)](z);
    CC = BITTST (R1, bitpos(XMTSERV)); /* test XMTSERV bit */
    if !CC jump XMTSERV_Status;
    W[P1 + LO(TWIO_INT_STAT)] = R1; /* clear status */
    SSYNC;

    /*********************************************************
    write byte into the transmit FIFO
    *********************************************************/
    R3 = B[P2++](Z);
    W[P1 + LO(TWIO_XMT_DATA8)] = R3;
  Loop_End1:  SSYNC;

/* check that master transfer has completed */
M_COMP1:
R1 = W[P1 + LO(TWIO_INT_STAT)](z);
CC = BITTST (R1, bitpos(MCOMP));
```

```
if !CC jump M_COMP1;
M_COMP1.END:W[P1 + LO(TWIO_INT_STAT)] = R1;

idle;
_main.end:
```

Slave Mode Setup

Listing 23-2 shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine _TWIO_ISR shown in Listing 23-3.

Listing 23-2. Slave Mode Setup

```
#include <defBF54x.h>
#include "startup.h"

#define file_size 254
#define SYSMMR_BASE 0xFFC00000
#define COREMMR_BASE 0xFFE00000

.GLOBAL _main;
.EXTERN _TWIO_ISR;

.section L1_data_b;
.BYTE TWIO_RX[file_size];
.BYTE TWIO_TX[file_size] = “transmit.dat”;

.section L1_code;
_main:

/*********************************************************
TWIO Slave Initialization subroutine
*********************************************************/
```

```
TWIO_SLAVE_INIT:

/*********************************************************
Enable the TWIO 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 = TWIO_ENA | 0xA (z);
W[P1 + LO(TWIO_CONTROL)] = R1;

/*********************************************************
Slave address
program the address to which this slave responds to.
this is an arbitrary 7-bit value
*********************************************************/
R1 = 0x5F;
W[P1 + LO(TWIO_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(TWIO_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(TWI0_FIFO_CTL)] = R1;

/*********************************************************
enable these signals to generate a TWI0 interrupt
*********************************************************/
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
W[P1 + LO(TWI0_INT_MASK)] = R1;

/*********************************************************
Enable the TWI0 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(TWI0_SLAVE_CTL)] = R1;
TWI0_SLAVE_INIT.END:

P2.H = HI(TWI0_RX);
P2.L = LO(TWI0_RX);

P4.H = HI(TWI0_TX);
P4.L = LO(TWI0_TX);
/*********************************************************
Remap the vector table pointer from the default __I10HANDLER
to the new _TWI0_ISR interrupt service routine
*********************************************************/
R1.H = HI(_TWI0_ISR);
R1.L = LO(_TWI0_ISR);
```

```
[P0 + LO(EVT10)] = R1; /* note that P0 points to the base of the
core MMR registers */

/*********************************************************
ENABLE TWI0 generate to interrupts at the system level
*********************************************************/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWI0));
[P1 + LO(SIC_IMASK)] = R1;

/*********************************************************
ENABLE TWI0 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 23-3. TWI0 Slave Interrupt Service Routine

```
/*********************************************************
 Function:  _TWI0_ISR
 Description:  This ISR is executed when the TWI0 controller
detects a slave initiated transfer. After an interrupt is ser-
viced, its corresponding bit is cleared in the TWI0_INT_STAT
register. This done by writing a 1 to the particular bit posi-
tion. All bits are write 1 to clear.
*********************************************************/
```

```
#include <defBF54x.h>

.GLOBAL _TWIO_ISR;

.section L1_code;
_TWIO_ISR:

/*********************************************************
read the source of the interrupt
*********************************************************/
R1 = W[P1 + LO(TWIO_INT_STAT)](z);

/*********************************************************
Slave Transfer Initiated
*********************************************************/
CC = BITTST(R1, BITPOS(SINIT));
if !CC JUMP RECEIVE;
R0 = SINIT (Z);
W[P1 + LO(TWIO_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;

/*********************************************************
Receive service
*********************************************************/
RECEIVE:
CC = BITTST(R1, BITPOS(RCVSERV));
if !CC JUMP TRANSMIT;
R0 = W[P1 + LO(TWIO_RCV_DATA8)] (Z); /* read data */
B[P2++] = R0 ; /* store bytes into a buffer pointed to by P2 */
R0 = RCVSERV(Z);
W[P1 + LO(TWIO_INT_STAT)] = R0; /*clear interrupt source bit */
ssync;
JUMP _TWIO_ISR.END; /* exit */
```

```
/*************************************************************
Transmit service
*************************************************************/
TRANSMIT:
CC = BITTST(R1, BITPOS(XMTSERV));
if !CC JUMP SlaveError;
R0 = B[P4++](Z);
W[P1 + LO(TWI0_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWI0_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWI0_ISR.END; /* exit */

/*************************************************************
slave transfer error
*************************************************************/
SlaveError:
CC = BITTST(R1, BITPOS(SERR));
if !CC SlaveOverflow;
R0 = SERR(Z);
W[P1 + if !CC  jump SlaveOverflow LO(TWI0_INT_STAT)] = R0; /*
clear interrupt source bit */
ssync;
JUMP _TWI0_ISR.END; /* exit */

/*************************************************************
slave overflow
*************************************************************/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWI0_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
```

```
JUMP _TWI0_ISR.END; /* exit */

/*********************************************************
 slave transfer complete
*********************************************************/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWI0_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWI0_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc. ... */
R0 = W[P1 + LO(TWI0_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 _TWI0_ISR.END;
R0 = W[P1 + LO(TWI0_RCV_DATA8)] (Z); /* read data */
B[P2++] = R0 ; /* store bytes into a buffer pointed to by P2 */

_TWI0_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.

24 SPORT CONTROLLERS

This chapter describes the processor's dual-channel synchronous serial ports (SPORTs) and includes the following sections:

- "Overview" on page 24-1
- "Interface Overview" on page 24-4
- "Description of Operation" on page 24-11
- "Functional Description" on page 24-28
- "SPORT Registers" on page 24-48
- "Programming Examples" on page 24-76

Overview

The ADSP-BF54x processor Blackfin processors feature four identical synchronous serial ports, called SPORTs. Unlike the SPI interface which is 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 I^2S 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.

All SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

The SPORTs can operate at up to 1/2 the system clock (SCLK) rate for an internally generated or external serial clock. Independent transmit and receive clocks provide greater flexibility for serial communications.

Features

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 data streams

- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 24-31 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 bit stream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 = (1023 – 128). Note the multichannel select registers and the `WSIZE` register control which subset of the 128 channels within the active region can be accessed.

Interface Overview

SPORT0, SPORT1, SPORT2, and SPORT3 provide an I/O interface to a wide variety of peripheral serial devices. SPORT0 is accessible through port C. SPORT1 is accessible through port D. SPORT2 and SPORT3 are both accessible through port A. For more information about port configuration see Chapter 9, “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`, `RFS1`, `RFS2`, and `RFS3` (corresponding to SPORT0, SPORT1, SPORT2, and SPORT3, respectively). In this chapter, LSB refers to least significant bit, and MSB refers to most significant bit.

Port A contains the SPORT2 and SPORT3 pins. Some of the SPORT2 and SPORT3 pins are multiplexed and can be used for other purposes if the entire SPORT2 and SPORT3 blocks or some of their signals are not required by an application. However, all pins default to the SPORT2 and SPORT3 modules settings after reset.

SPORT0 resides in port C. Its secondary data pins are shared with MXVR. The `PORTC_MUX` register controls whether the secondary SPORT0 data lines are enabled. By default, all port C pins are configured in GPIO mode. Writing to `PORTC_FER` enables peripheral functionality. For more information see Chapter 9, “General-Purpose Ports”.

SPORT1 resides in port D. Its signals are shared with the PPI and HDMA. The PORTD_MUX register controls whether the SPORT1 lines are enabled. By default, all port D pins are configured in GPIO mode. Writing to PORTD_FER enables peripheral functionality. For more information see Chapter 9, "General-Purpose Ports".

The secondary data pins of SPORT2 and SPORT3 are multiplexed with general-purpose timers. The PORTA_MUX register determines whether general-purpose timer functionality is enabled. The remaining SPORT2 and SPORT3 signals aren't multiplexed, but they can be used as GPIO pins as dictated by the PORTA_FER register. For more information see Chapter 9, "General-Purpose Ports".

On DMAC1, 32-bit DMA mode is not supported for SPORT2 or SPORT3. The data word lengths for SPORT2 and SPORT3 may, however, still be set to 32 bits.

Figure 24-1 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 through the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the transmit 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 24-1 shows the signals for each SPORT.

Table 24-1. 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, 1, 2, or 3 (corresponding to SPORT0, SPORT1, SPORT2, and SPORT3).

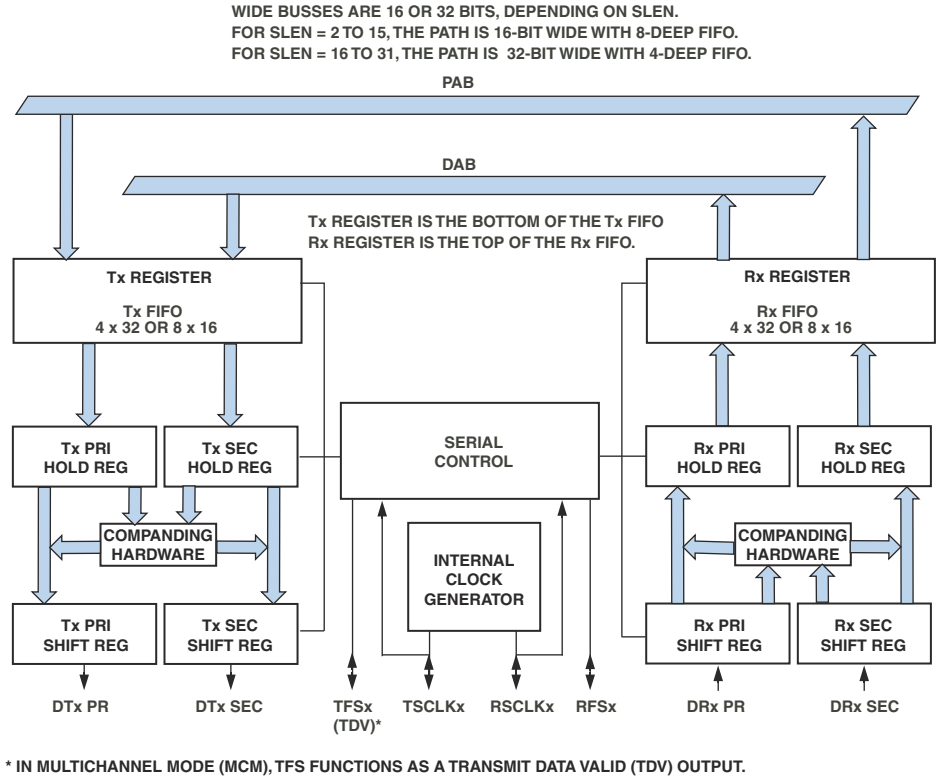


Figure 24-1. SPORT Block Diagram

Blackfin SPORTs are designed such that I^2S master mode, LRCLK, is held at the last driven logic level and does not transition, to provide an edge, after the final data word is driven out. Therefore, while transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a Blackfin SPORT.

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 *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 24-2 shows a possible port connection for the SPORTs. Note that serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial pairs C and D, E and F, and G and H. SPORT1 is Multichannel Mode. In Multichannel mode, TFS functions as a transmit data valid (TDV) output. Although shown as an external connection, the TSCLK/RSCLK connection is internal in multichannel mode. See "Multichannel Operation" on page 24-17 for details.

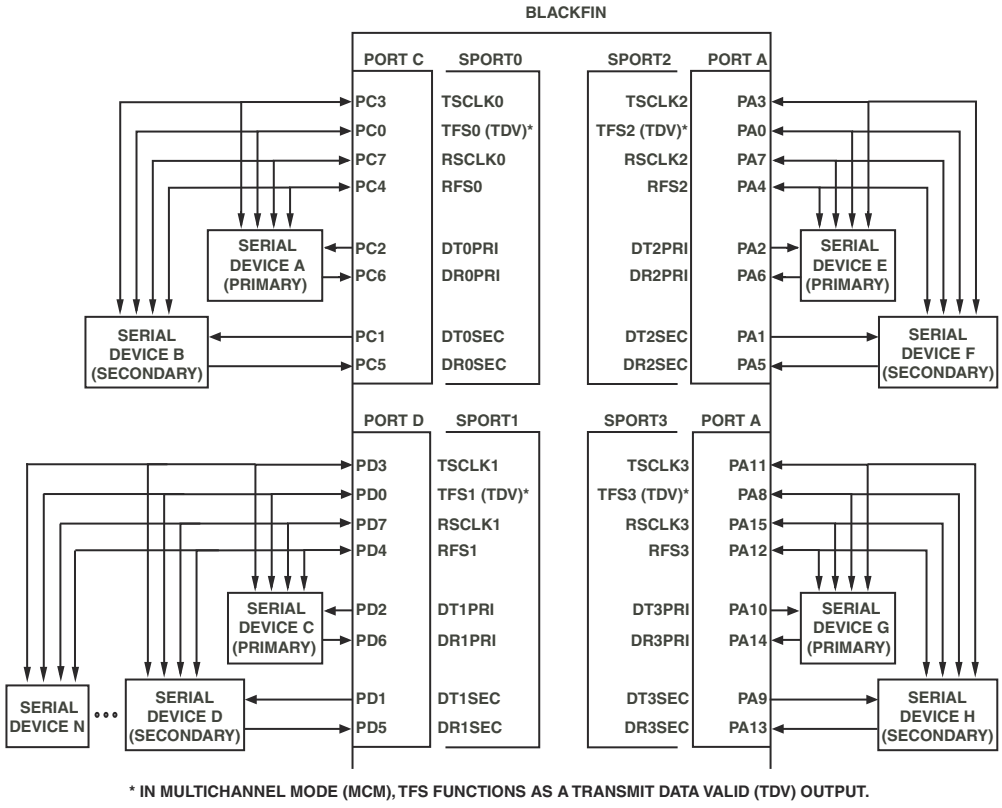


Figure 24-2. SPORT3–0 Example Connections

Figure 24-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

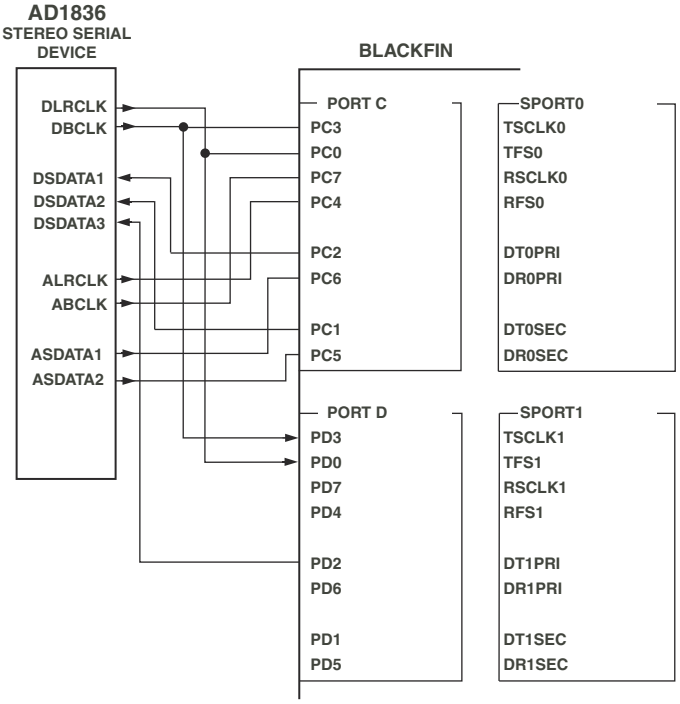


Figure 24-3. SPORT1–0 Example Stereo Serial Connection

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

The following sections describe the operation of the SPORT controllers.

SPORT Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT's SPORTx_TX register readies the SPORT for transmission. The TFSx signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORTx_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORTx_TCR1 register. Each bit is shifted out on the driving edge of TSCLKx. The driving edge of TSCLKx can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word is 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 more information see Chapter 7, "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 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, disable the SPORT.

Note that disabling a SPORT through 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.

When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. Each SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TCR1 and/or RSPEN in SPORTx_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the "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^{2}S 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^{2}S 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 24-4 on page 24-13 shows timing diagrams for stereo serial mode transmit operation.

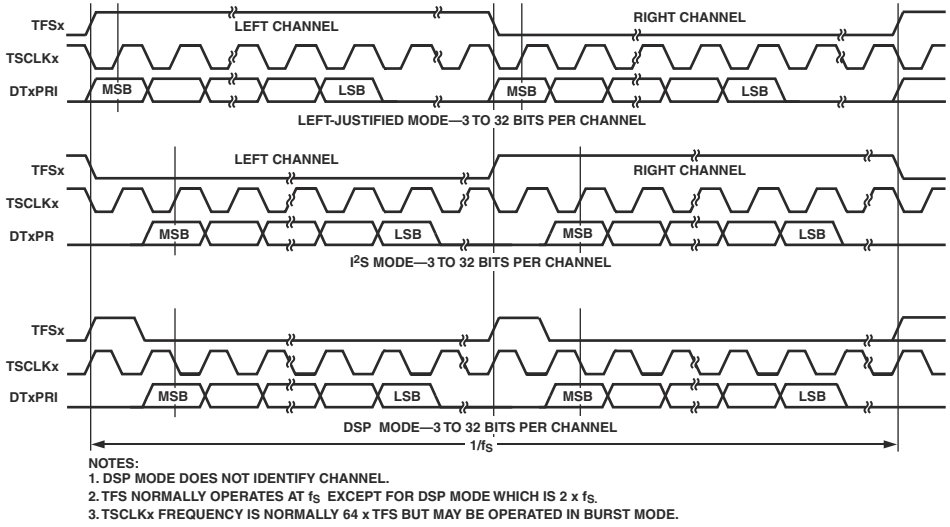


Figure 24-4. SPORT Stereo Serial Modes, Transmit

Figure 24-5 on page 24-14 shows timing diagrams for stereo serial mode receive operation.

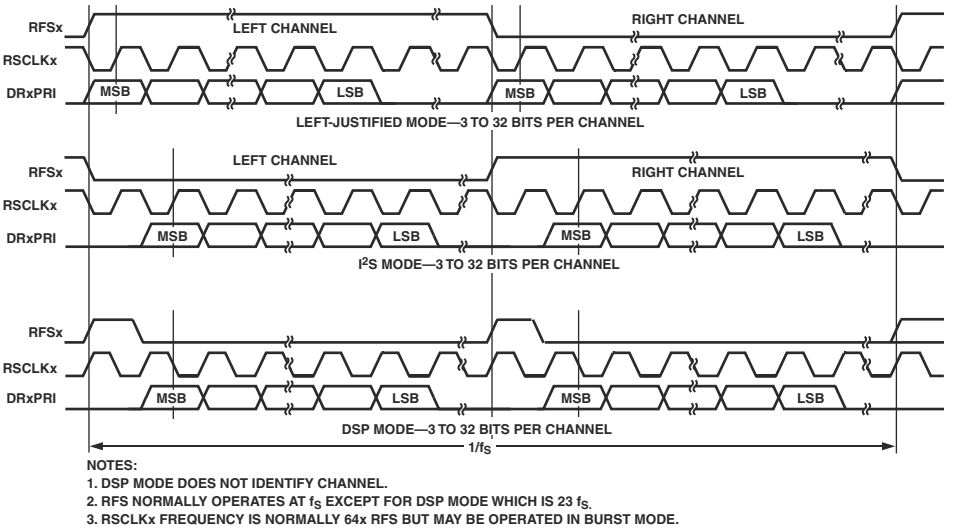


Figure 24-5. SPORT Stereo Serial Modes, Receive

Blackfin SPORTs are designed such that, in I^2S master mode, LRCLK is held at the last driven logic level and does not transition, to provide an edge, after the final data word is driven out. Therefore, while transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a Blackfin SPORT.

Table 24-2 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 trans-

mit 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 24-2. 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
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 24-2 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I²S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 24-28) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a 'right' channel. Thus, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the 'right' channel, thus implying that it is low for 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 I²S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 24-3 on page 24-10, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

Multichannel Operation

The SPORTs offer a multichannel mode of operation which allows the SPORT to communicate in a Time-Division-Multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bit stream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024 total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTxPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORTx_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTxSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORTx_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

In multichannel mode, RSCLKx can either be provided externally or generated internally by the SPORT, and it is used for both transmit and receive functions. Leave TSCLKx disconnected if the SPORT is used only in multichannel mode. If RSCLKx is externally or internally provided, it will be internally distributed to both the receiver and transmitter circuitry.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORTx_TX or SPORTx_RX operation for multichannel mode. This is especially important in "DMA data unpacked mode," since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORTx_MCMC2) must be enabled prior to enabling SPORTx_TX or SPORTx_RX operation. When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 24-6 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
- RFSx signals start of frame
- TFSx 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 24-42 for more examples.

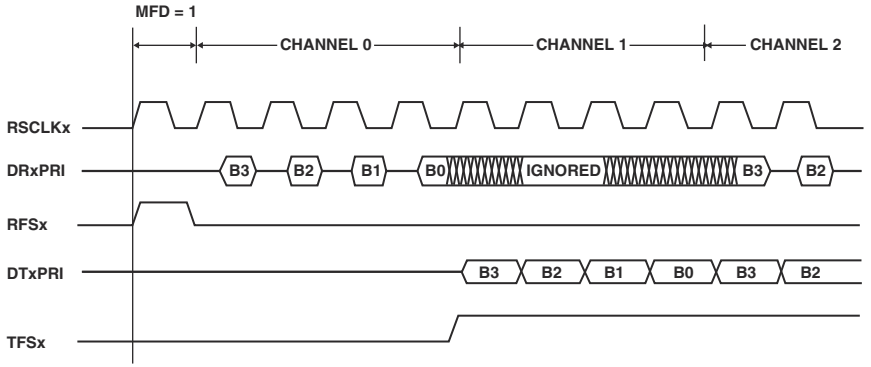


Figure 24-6. Multichannel Operation

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 24-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 24-3. 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
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 RFSx signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFSx as a frame sync. This is true whether RFSx is generated internally or externally. The RFSx signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFSx indicates the beginning of the channel 0 data word.

Since RFSx 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, RFSx timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFSx signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFSx signal serves as an output-enabled signal for the data transmit pin. The SPORT drives TFSx in multichannel mode whether or not ITFS is cleared. The TFSx 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 TFSx pin indicates an active channel.

Once the initial RFSx is received, and a frame transfer has started, all other RFSx signals are ignored by the SPORT until the complete frame is transferred.

If MFD > 0, the RFSx may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFSx signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFSx signals required. Therefore, internally generated frame syncs are always data independent.

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 24-7.

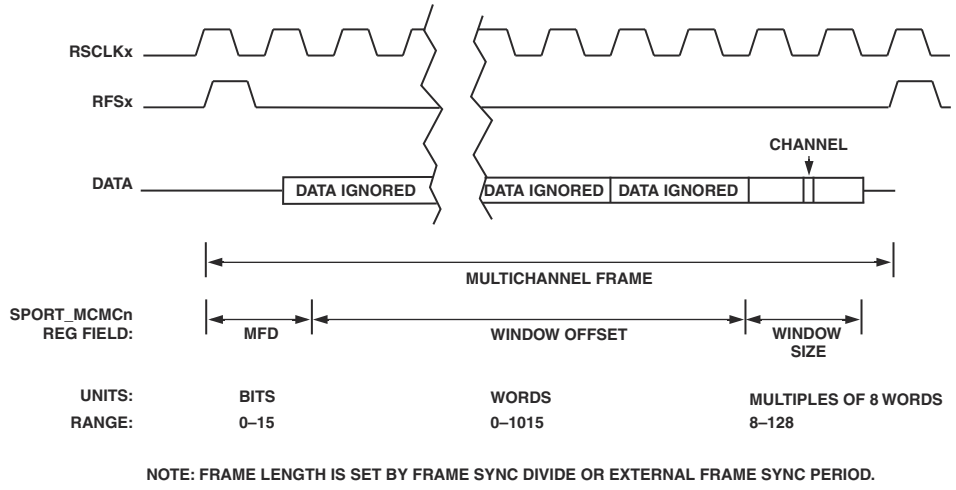


Figure 24-7. Relationships for Multichannel Parameters

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, which is the equivalent of the late frame sync mode. MFD>0 corresponds to the early frame sync mode. There a new frame sync may occur before data from the last frame is received, because blocks of data occur back-to-back. The maximum value allowed for MFD is 15.

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.

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 the data is transmitted on the same edge that the TFSx is generated (FSDR = 0). For example, a positive edge on TFSx causes data to be transmitted on the positive edge of the TSCLKx—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a 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 24-8.

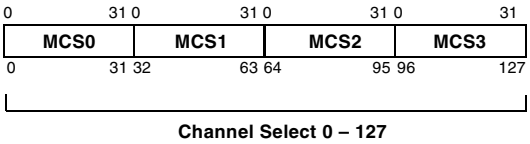


Figure 24-8. Multichannel Select Registers

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 data stream. Clearing the bit in the SPORTx_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORTx_MRCSn register causes the SPORT to receive the word in that channel's position of the data stream; the received word is loaded into the SPORTx_RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the TDTYPE field in the SPORTx_TCR1 register and the RDTYPE field in the SPORTx_RCR1 register, and applies to all active channels. (See "Companding" on page 24-31 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 (2M bps data) and HMVIP (8M bps 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 = b#00 chooses non-divide or bypass mode (H.100-compatible),

MCCRM = b#10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = b#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 TSCLKx or RSCLKx frequency takes effect at the start of the drive edge of TSCLKx or RSCLKx that follows the next leading edge of TFSx or RFSx.

When an internal frame sync is selected (ITFS = 1 in the SPORTx_TCR1 register or IRFS = 1 in the SPORTx_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORTx_TCLKDIV or SPORTx_RCLKDIV has changed. The second frame sync will cause the update.

The SPORTx_TFSDIV and SPORTx_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFSx or RFSx pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

# of transmit serial clocks between frame sync assertions = TFSDIV + 1

# of receive serial clocks between frame sync assertions = RFSDIV + 1

Use the following equations to determine the correct value of 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 *ADSP-BF542/544/547/548/549 Embedded Processor 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 24-4.

Table 24-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORTx_TCR1 Data Formatting	SPORTx_RCR1 Data Formatting
b#00	Normal operation	Zero fill
b#01	Reserved	Sign extend
b#10	Compand using μ-law	Compand using μ-law
b#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 (TSCLKx) and a receive clock signal (RSCLKx). The clock signals are configured by the TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers. Serial clock frequency is configured in the SPORTx_TCLKDIV and SPORTx_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORTx_TCR1 configuration register and the IRCLK bit in the SPORTx_RCR1 configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the processor, and the TSCLKx or RSCLKx pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLKx or RSCLKx pins, and the serial clock divisors in the SPORTx_TCLKDIV/SPORTx_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the system clock or with each other. The system clock must have a higher frequency than RSCLKx and TSCLKx.

When the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the processor data sheet for details.

The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFSx (transmit frame sync) and RFSx (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 TFSx and RFSx 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 24-9 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 24-42 for more timing examples.

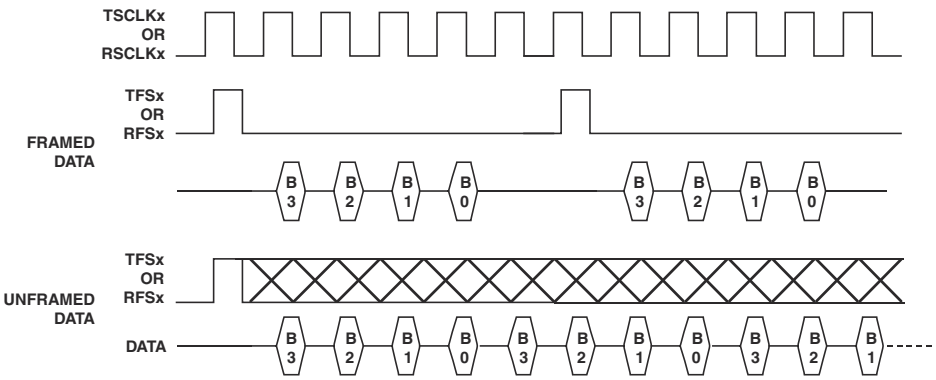


Figure 24-9. Framed Versus Unframed Data

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 TFSx pin or RFSx 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 TFSx pin or RFSx 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 24-10, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

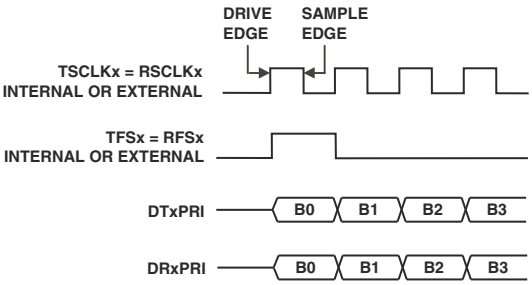


Figure 24-10. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 24-11, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

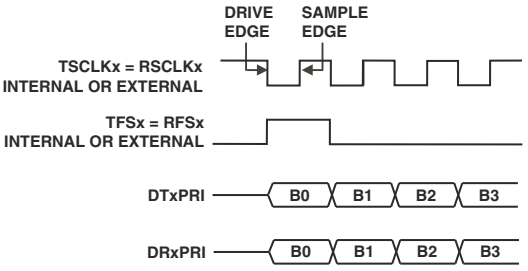


Figure 24-11. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

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 is 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 $\geq$ 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 24-12 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 24-42 for more examples.

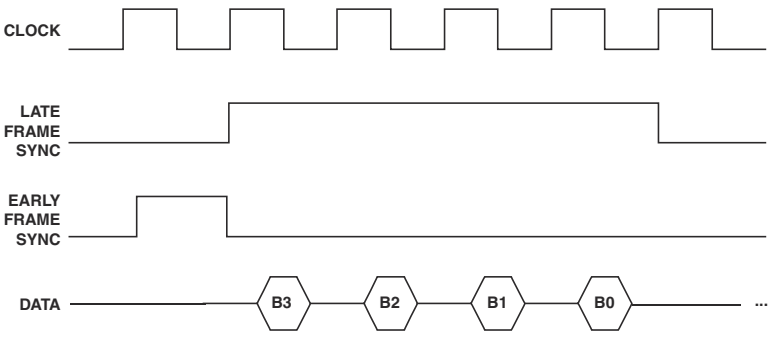


Figure 24-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFSx) 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 TFSx signal, with or without new data. The DITFS bit of the SPORTx_TCR1 register configures this option.

When DITFS = 0, the internally generated TFSx is only output when a new data word is loaded into the SPORTx_TX buffer. The next TFSx 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 TFSx 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 TFSx. 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 TFSx occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFSx 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 more information see Chapter 7, “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, SPORTx_TX)
- Writing a read-only register (for example, SPORTx_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 24-33, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 24-38, and "Frame Syncs in Multichannel Mode" on page 24-20). 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 *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing signals are active high (LRFS = 0 and LTFS = 0).

Figure 24-13 through Figure 24-18 show framing for receiving data.

In Figure 24-13 and Figure 24-14, the normal framing mode is shown for non-continuous data (any number of TSCLKx or RSCLKx cycles between words) and continuous data (no TSCLKx or SRSCLKx cycles between words).

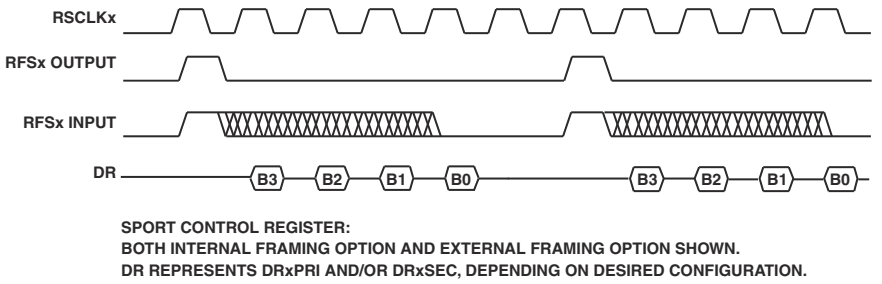


Figure 24-13. SPORT Receive, Normal Framing

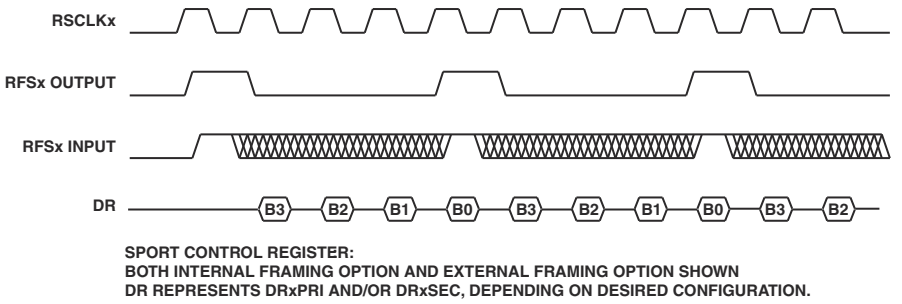


Figure 24-14. SPORT Continuous Receive, Normal Framing

Figure 24-15 and Figure 24-16 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 `RFSx` for the other SPORT channel.

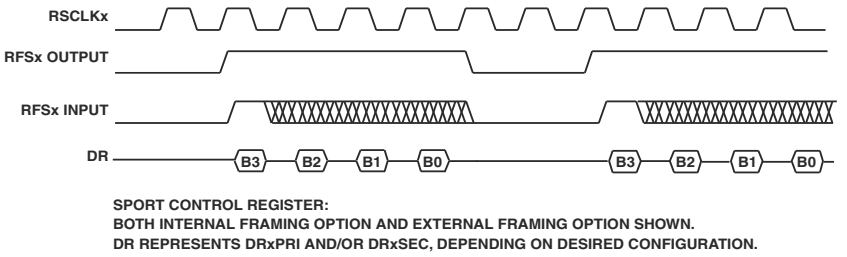


Figure 24-15. SPORT Receive, Alternate Framing

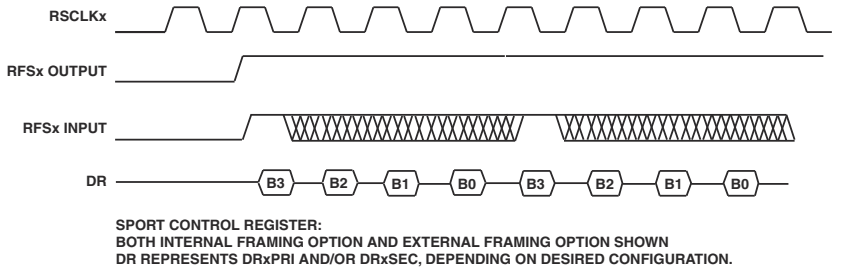


Figure 24-16. SPORT Continuous Receive, Alternate Framing

Figure 24-17 and Figure 24-18 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 RSCLKx before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

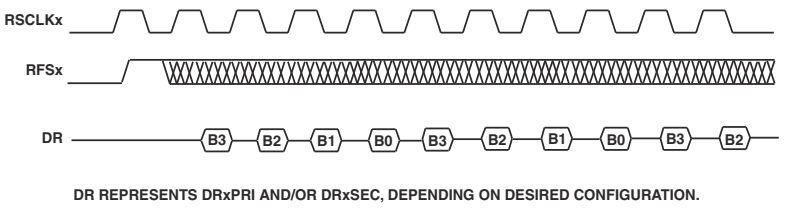


Figure 24-17. SPORT Receive, Unframed Mode, Normal Framing

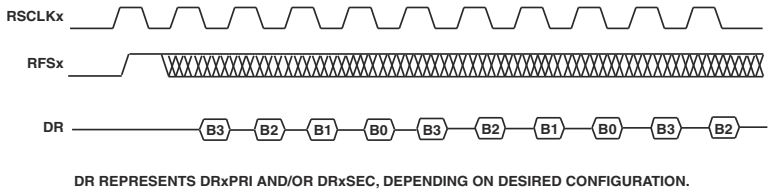


Figure 24-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 24-19 through Figure 24-24 show framing for transmitting data and are very similar to Figure 24-13 through Figure 24-18.

In Figure 24-19 and Figure 24-20, the normal framing mode is shown for non-continuous data (any number of TSCLKx cycles between words) and continuous data (no TSCLKx cycles between words). Figure 24-21 and Figure 24-22 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFSx output meets the RFSx input timing requirement.

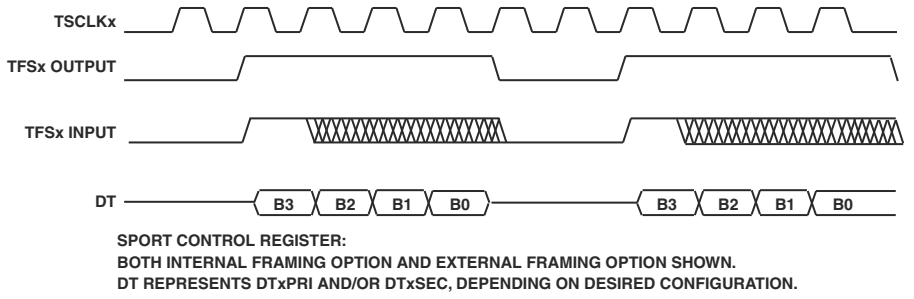


Figure 24-19. SPORT Transmit, Normal Framing

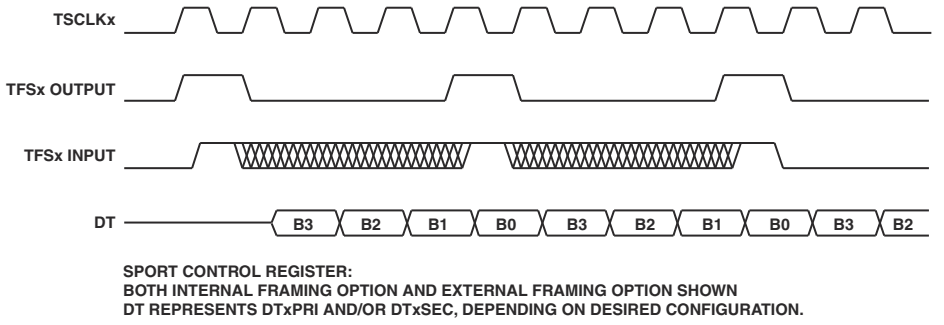


Figure 24-20. SPORT Continuous Transmit, Normal Framing

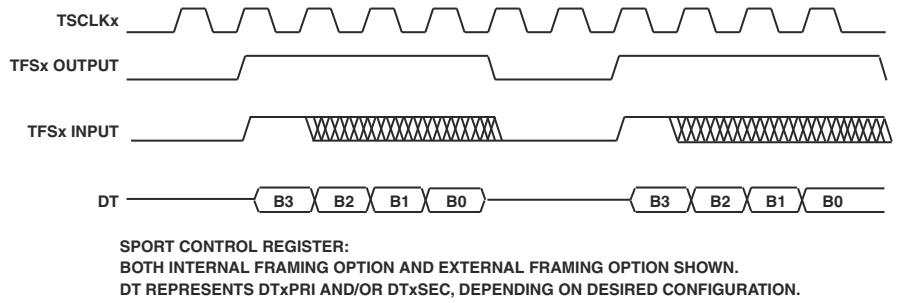


Figure 24-21. SPORT Transmit, Alternate Framing

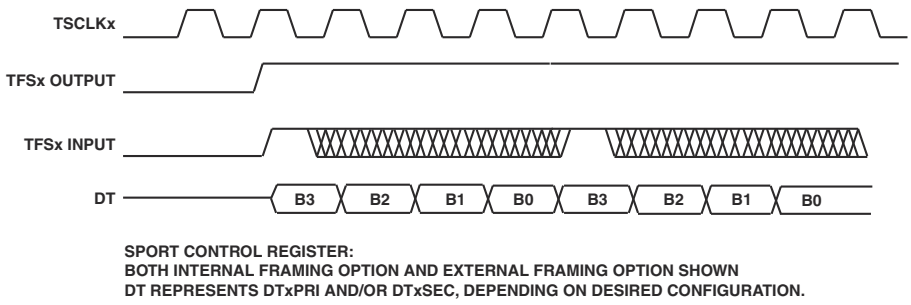


Figure 24-22. SPORT Continuous Transmit, Alternate Framing

Figure 24-23 and Figure 24-24 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 TSCLKx before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

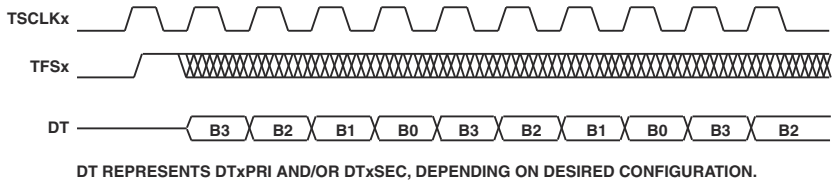


Figure 24-23. SPORT Transmit, Unframed Mode, Normal Framing

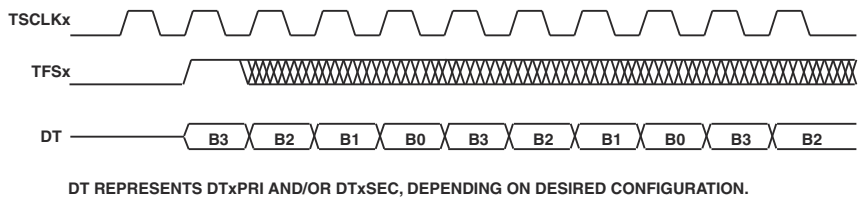


Figure 24-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. Table 24-5 provides an overview of the available control registers.

Table 24-5. SPORT Registers

Register Name	Description	Notes
SPORTx_TCR1	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51	Bits [15:1] can only be written if bit 0 = 0
SPORTx_TCR2	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51	
SPORTx_TCLKDIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68	Ignored if external SPORT clock mode is selected
SPORTx_TFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69	Ignored if external frame sync mode is selected
SPORTx_TX	"Transmit Data (SPORTx_TX) Register" on page 24-61	
SPORTx_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56	Bits [15:1] can only be written if bit 0 = 0
SPORTx_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56	
SPORTx_RCLKDIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68	Ignored if external SPORT clock mode is selected
SPORTx_RFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69	Ignored if external frame sync mode is selected
SPORTx_RX	"Receive Data (SPORTx_RX) Register" on page 24-64	
SPORTx_STAT	"SPORT Status (SPORTx_STAT) Register" on page 24-66	

Table 24-5. SPORT Registers (Cont'd)

Register Name	Description	Notes
SPORTx_MCMC1	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70	Configure this register before enabling the SPORT
SPORTx_MCMC2	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70	Configure this register before enabling the SPORT
SPORTx_MRCSn	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72	Select or deselect channels in a multichannel frame
SPORTx_MTCSn	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74	Select or deselect channels in a multichannel frame
SPORTx_CHNL	"Current Channel (SPORTx_CHNL) Register" on page 24-71	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.

Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers

The main control registers for the transmit portion of each SPORT are registers SPORTx_TCR1 and SPORTx_TCR2, shown in Figure 24-25 and Figure 24-26.

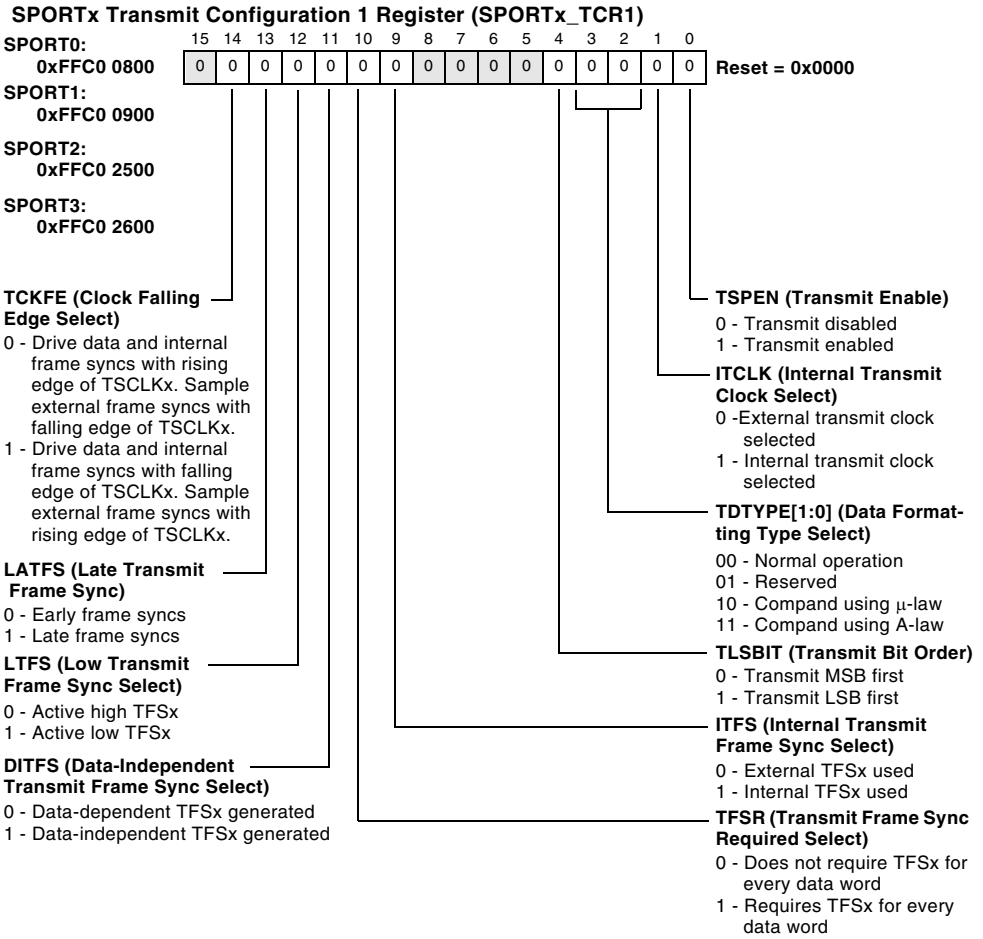


Figure 24-25. SPORTx Transmit Configuration 1 Register

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) ;  /* SPORT disabled, SPORTx_TCR1
                                   still equal to 0x0000 */
```

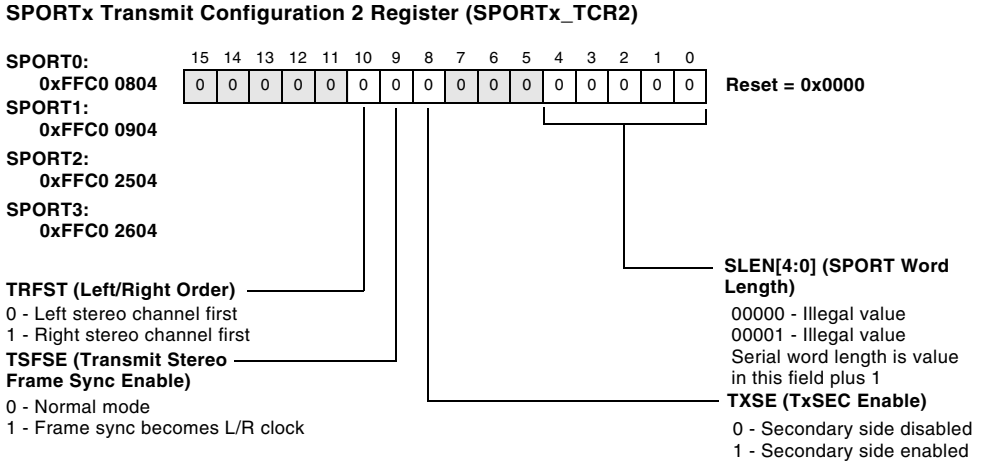


Figure 24-26. SPORTx Transmit Configuration 2 Register

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, TSCLKx, 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 TSCLKx 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 through 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 TFSx (if set) or an external TFSx (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 TFSx (sync at selected interval) or a data-dependent TFSx (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 is loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORTx_TX data register is 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 TFSx (if set) or active high TFSx (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 TSCLKx 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.

- **Transmit secondary 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 24-27 and Figure 24-28.

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 */
```

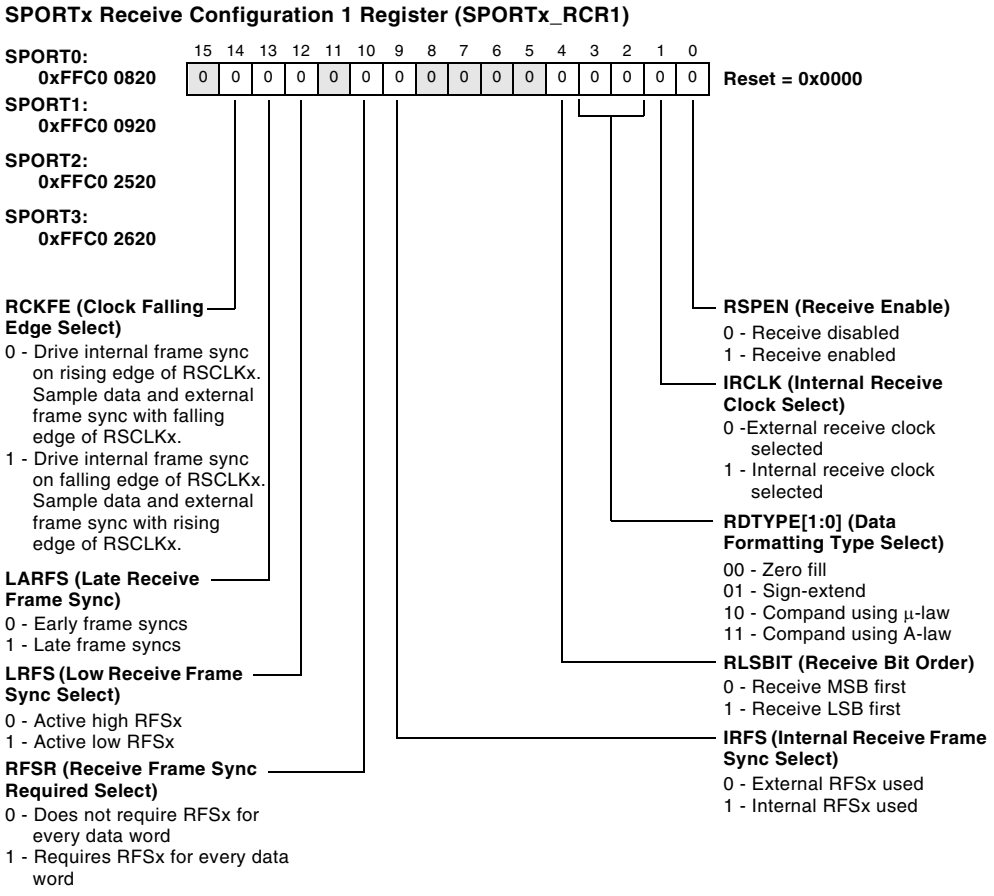


Figure 24-27. SPORTx Receive Configuration 1 Register

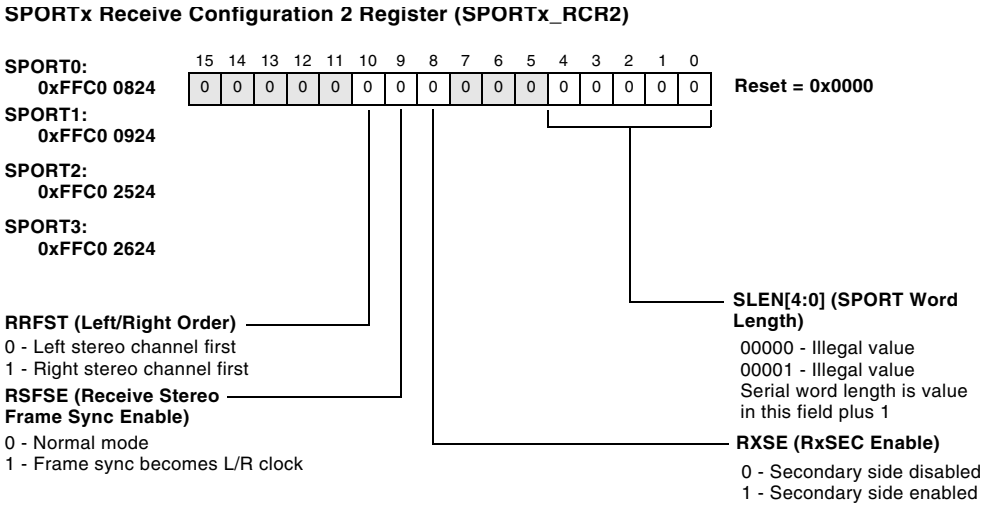


Figure 24-28. SPORTx Receive Configuration 2 Register

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 RFSx (if set) or an external RFSx (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 RFSx (if set) or active high RFSx (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 RSCLKx 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.

Transmit Data (SPORTx_TX) Register

The SPORTx transmit data register (SPORTx_TX) is a write-only register. Reads produce a Peripheral Access Bus (PAB) error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length <= 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 24-29. The SPORTx_TX register is shown in Figure 24-30.

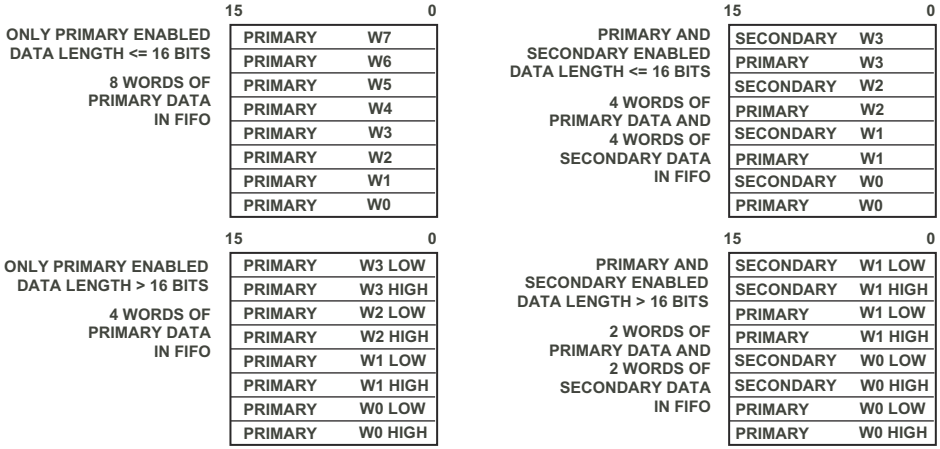


Figure 24-29. SPORT Transmit FIFO Data Ordering

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 more information see Chapter 7, “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 is 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 TSPEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORTx_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORTx_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT status register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORTx status register are updated upon writes from the core processor, even when the SPORT is disabled.

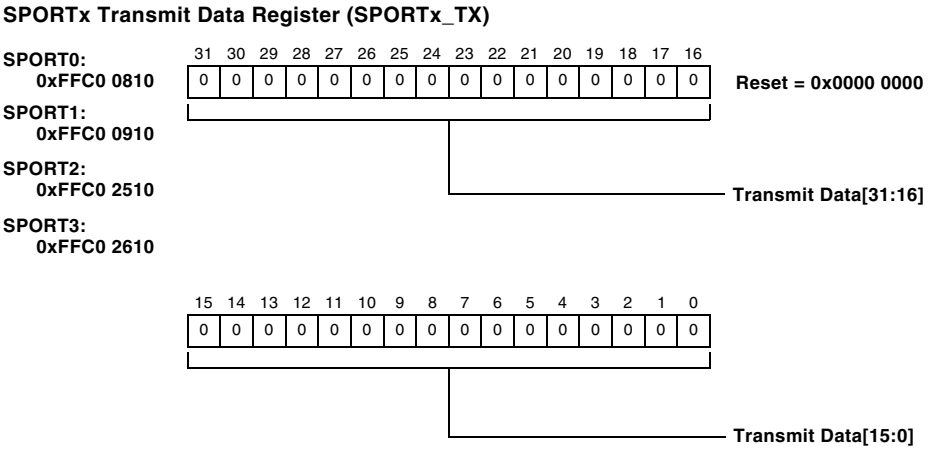


Figure 24-30. SPORTx Transmit Data Register

Receive Data (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 24-31. The SPORTx_RX register is shown in Figure 24-32.

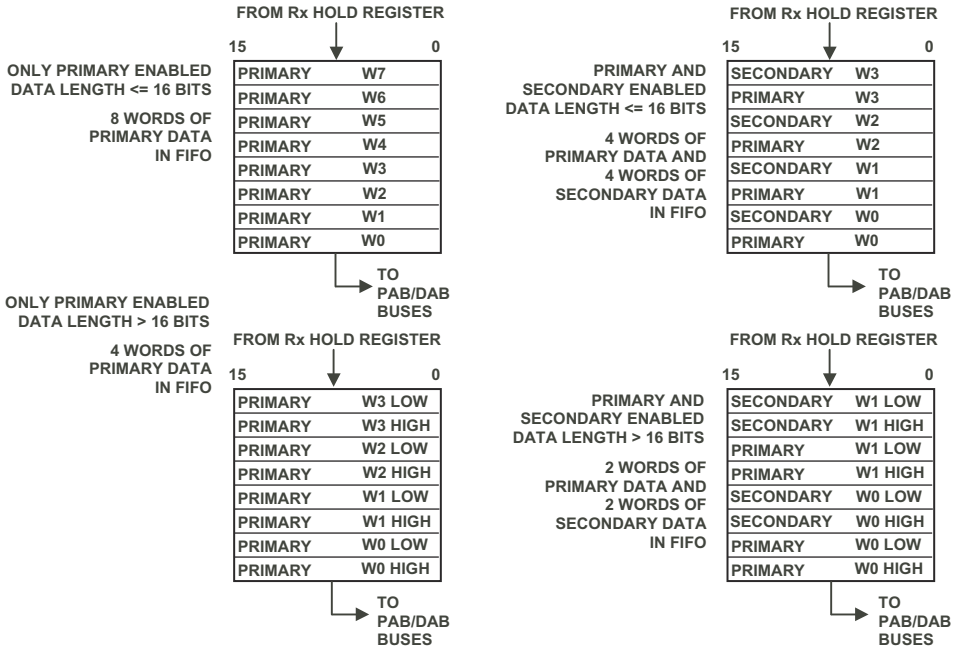


Figure 24-31. SPORT Receive FIFO Data Ordering

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.

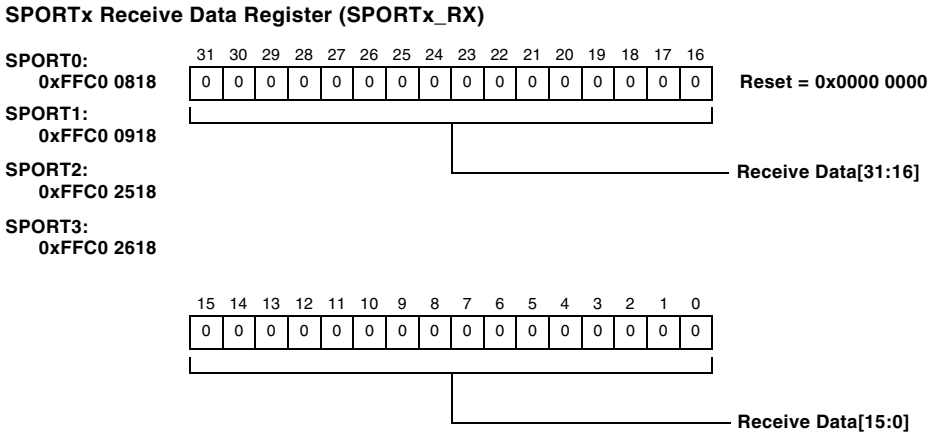


Figure 24-32. SPORTx Receive Data Register

SPORT Status (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 24-33.

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 TFSx signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFSx 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 TSPEN = 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 TSPEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT status register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing RSPEN = 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 RSPEN = 0. The RUVF bit is updated even when the SPORT is disabled.

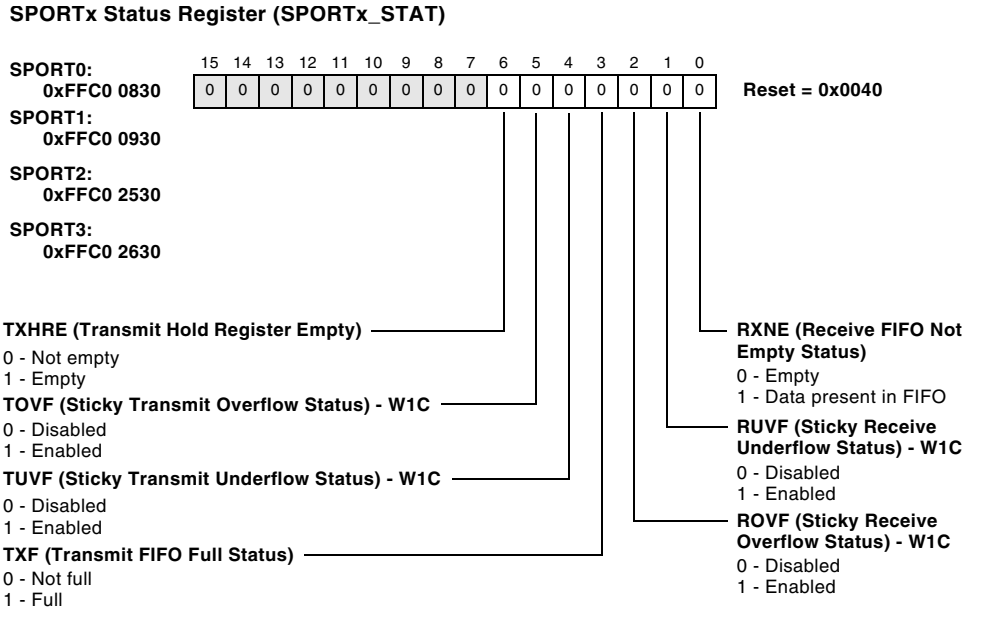


Figure 24-33. SPORTx Status Register

Serial Clock Divider (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 divider registers. The SPORTx transmit serial clock divider register, SPORTx_TCLKDIV is shown in Figure 24-34, and the SPORTx receive serial clock divider register, SPORTx_RCLKDIV is shown in Figure 24-35.

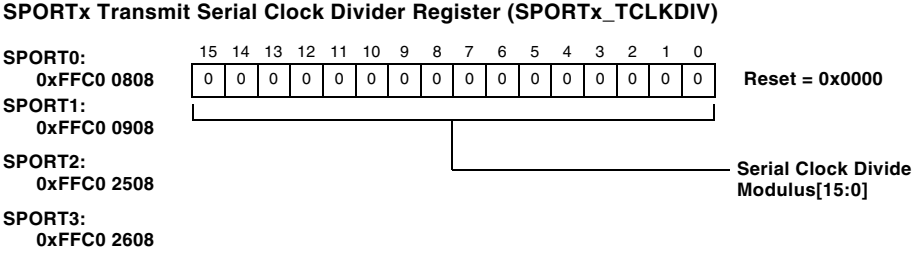


Figure 24-34. SPORTx Transmit Serial Clock Divider Register

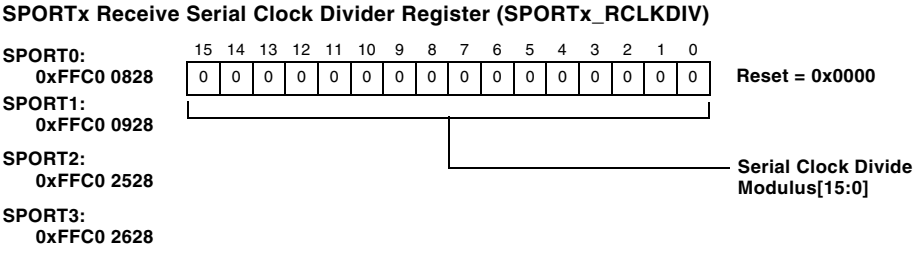


Figure 24-35. SPORTx Receive Serial Clock Divider Register

Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers

The 16-bit SPORTx transmit frame sync divider register (SPORTx_TFSDIV) and the SPORTx receive frame sync divider register (SPORTx_RFSDIV) specify how many transmit or receive clock cycles are counted before generating a TFSx or RFSx pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks. These registers are shown in Figure 24-36 and Figure 24-37.

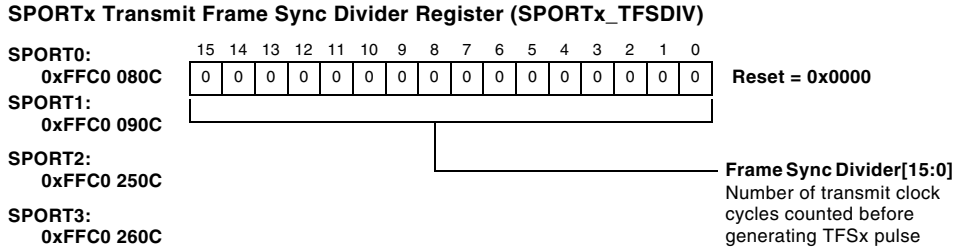


Figure 24-36. SPORTx Transmit Frame Sync Divider Register

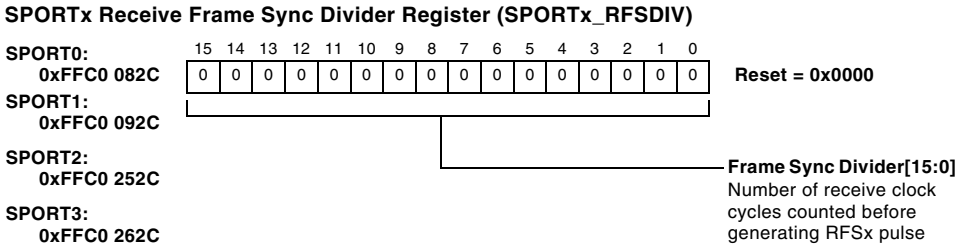


Figure 24-37. SPORTx Receive Frame Sync Divider Register

Multichannel Configuration (SPORTx_MCMCn) Registers

There are two SPORTx multichannel configuration registers (SPORTx_MCMCn) for each SPORT, shown in Figure 24-38 and Figure 24-39. The SPORTx_MCMCn registers are used to configure the multichannel operation of the SPORT.

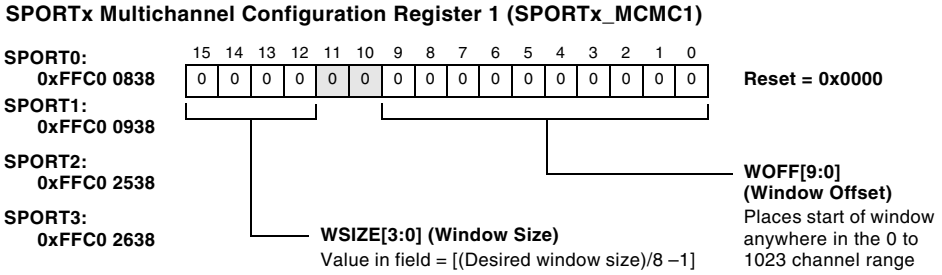


Figure 24-38. SPORTx Multichannel Configuration Register 1

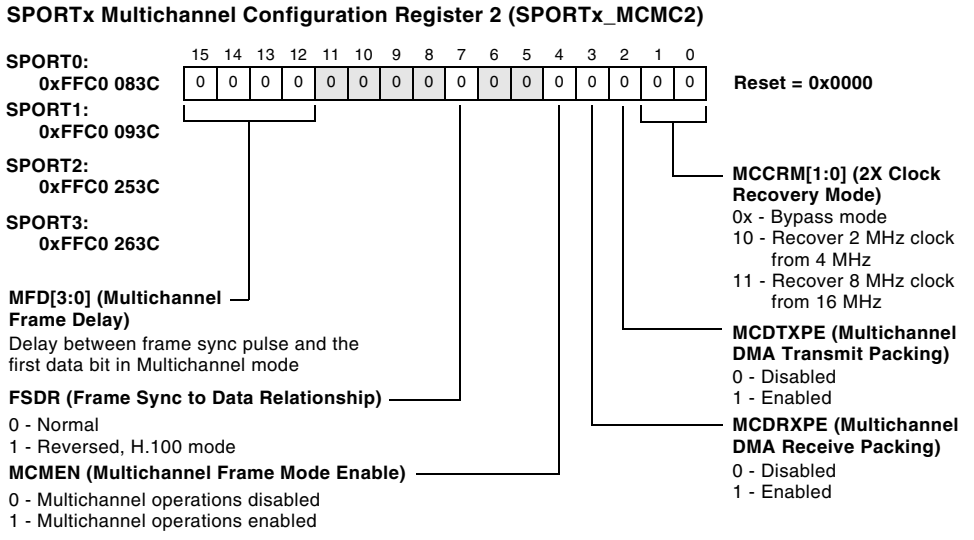


Figure 24-39. SPORTx Multichannel Configuration Register 2

Current Channel (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 `RSCLKx` and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind.

The SPORTx_CHNL register is shown on Figure 24-40.

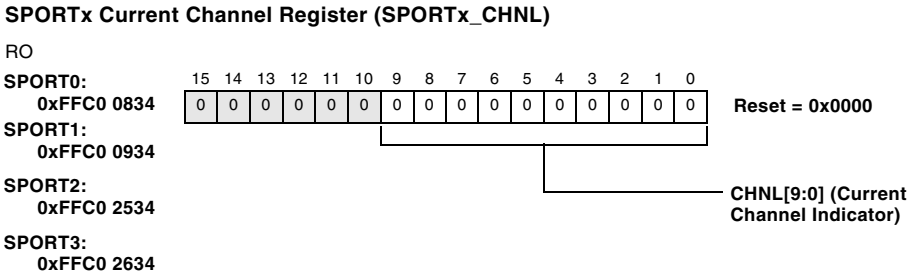


Figure 24-40. SPORTx Current Channel Register

Multichannel Selection Receive (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 24-41) 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 data stream; 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. Table 24-6 lists memory-mapped addresses for all SPORTx_MRCSn registers.

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 24-6.

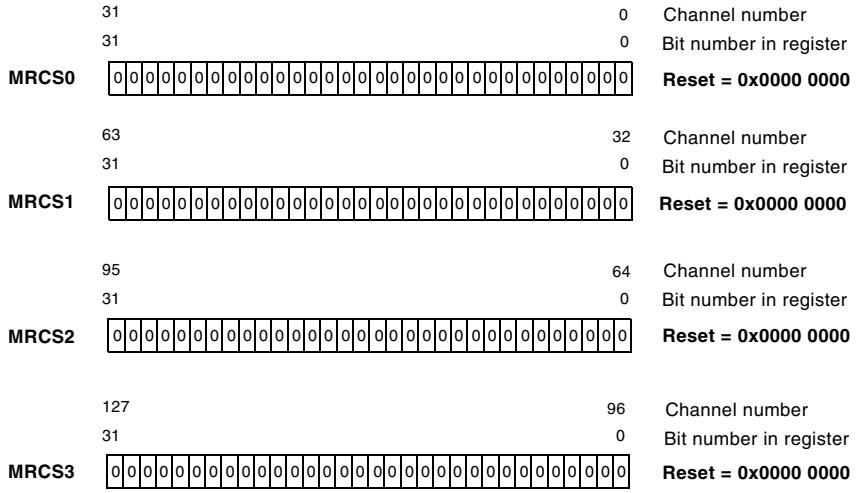


Figure 24-41. SPORTx Multichannel Receive Select Registers

Table 24-6. SPORTx Multichannel Receive Select Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address	Register Name	Memory-Mapped Address
SPORT0_MRCS0	0xFFC0 0850	SPORT2_MRCS0	0xFFC0 2550
SPORT0_MRCS1	0xFFC0 0854	SPORT2_MRCS1	0xFFC0 2554
SPORT0_MRCS2	0xFFC0 0858	SPORT2_MRCS2	0xFFC0 2558
SPORT0_MRCS3	0xFFC0 085C	SPORT2_MRCS3	0xFFC0 255C
SPORT1_MRCS0	0xFFC0 0950	SPORT3_MRCS0	0xFFC0 2650
SPORT1_MRCS1	0xFFC0 0954	SPORT3_MRCS1	0xFFC0 2654
SPORT1_MRCS2	0xFFC0 0958	SPORT3_MRCS2	0xFFC0 2658
SPORT1_MRCS3	0xFFC0 095C	SPORT3_MRCS3	0xFFC0 265C

Multichannel Selection Transmit (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 24-42) 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 data stream. 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 24-7.

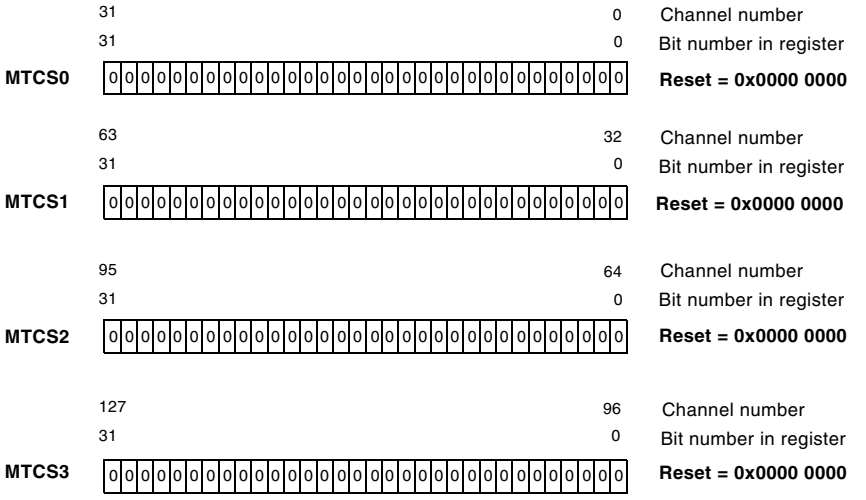


Figure 24-42. SPORTx Multichannel Transmit Select Registers

Table 24-7. SPORTx Multichannel Transmit Select Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address	Register Name	Memory-Mapped Address
SPORT0_MTCS0	0xFFC0 0840	SPORT2_MTCS0	0xFFC0 2540
SPORT0_MTCS1	0xFFC0 0844	SPORT2_MTCS1	0xFFC0 2544
SPORT0_MTCS2	0xFFC0 0848	SPORT2_MTCS2	0xFFC0 2548
SPORT0_MTCS3	0xFFC0 084C	SPORT2_MTCS3	0xFFC0 254C
SPORT1_MTCS0	0xFFC0 0940	SPORT3_MTCS0	0xFFC0 2640
SPORT1_MTCS1	0xFFC0 0944	SPORT3_MTCS1	0xFFC0 2644
SPORT1_MTCS2	0xFFC0 0948	SPORT3_MTCS2	0xFFC0 2648
SPORT1_MTCS3	0xFFC0 094C	SPORT3_MTCS3	0xFFC0 264C

Programming Examples

Listing 24-1 through Listing 24-4 on page 24-82 show how a SPORT is used in conjunction with the DMA controller.

Since serial ports are usually employed for high-speed, continuous serial transfers, this example shows an auto-buffered, repeated DMA transfer.

While there are many possible configurations, this 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. All examples assume core writes to `PORTx_FER` and `PORTx_MUX` have been made to properly configure port pins associated with the SPORT module.

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 24-1. 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 (= 0 to 65535) */
  W[P0 + (SPORT0_TCLKDIV - SPORT0_TCR1)] = R1;  /* TCK divider
                                                   register */
/* number of Bitclocks between FrameSyncs -1 (= SPORT_SLEN to
65535) */
  R1 = SPORT_TFSDIV_CONFIG;
  W[P0 + (SPORT0_TFSDIV - SPORT0_TCR1)] = R1;   /* TFSDIV
                                                   register */

  /* 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 register
*/

/* 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 external
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 (DMA channel 0) and for transmit (DMA channel 1) 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. For more information see Chapter 7, “Direct Memory Access”.

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 24-2. DMA Initialization

```
Program_DMA_Controller:

/* Receiver (DMA channel 0) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA0_CONFIG);
  P0.h = hi(DMA0_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 + (DMA0_X_COUNT - DMA0_CONFIG)] = R1;  /* X_count register
*/
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
```

```
  W[P0 + (DMA0_X_MODIFY - DMA0_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 + (DMA0_START_ADDR - DMA0_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 0channel 1) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA1_CONFIG);
  P0.h = hi(DMA1_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 + (DMA1_X_COUNT - DMA1_CONFIG)] = R1;  /* X_count register
*/
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA1_X_MODIFY - DMA1_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 + (DMA1_START_ADDR - DMA1_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 24-3. Servicing an Interrupt

```
RECEIVE_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA0_IRQ_STATUS);
  P0.l = lo(DMA0_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(DMA1_IRQ_STATUS);
  P0.l = lo(DMA1_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 24-4. 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;
```

25 UART PORT CONTROLLERS

This chapter describes the universal asynchronous receiver/transmitter (UART) modules and includes the following sections:

- "Overview" on page 25-1
- "Interface Overview" on page 25-3
- "Description of Operation" on page 25-6
- "Programming Model" on page 25-22
- "UART Registers" on page 25-26
- "Programming Examples" on page 25-51

Overview

The ADSP-BF54x processor Blackfin processors feature multiple separate and identical UART modules.

ADSP-BF548 and ADSP-BF549 processors feature four UARTs, referred to as UART0, UART1, UART2, and UART3. UART2 is not present on ADSP-BF542 and ADSP-BF544 devices.

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
- Additional 4-stage receive FIFO with programmable threshold interrupt
- Flexible transmit and receive interrupt timings
- 3 interrupt outputs for reception, transmission, and status
- Independent DMA operation for receive and transmit
- Programmable automatic RTS/CTS hardware flow control on UART1 and UART3
- False start bit detection
- SIR IrDA operation mode
- Internal loop back
- Improved bit rate granularity

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 25-1 shows a simplified block diagram of one UARTx module and how it interconnects to the Blackfin architecture and to the outside world.

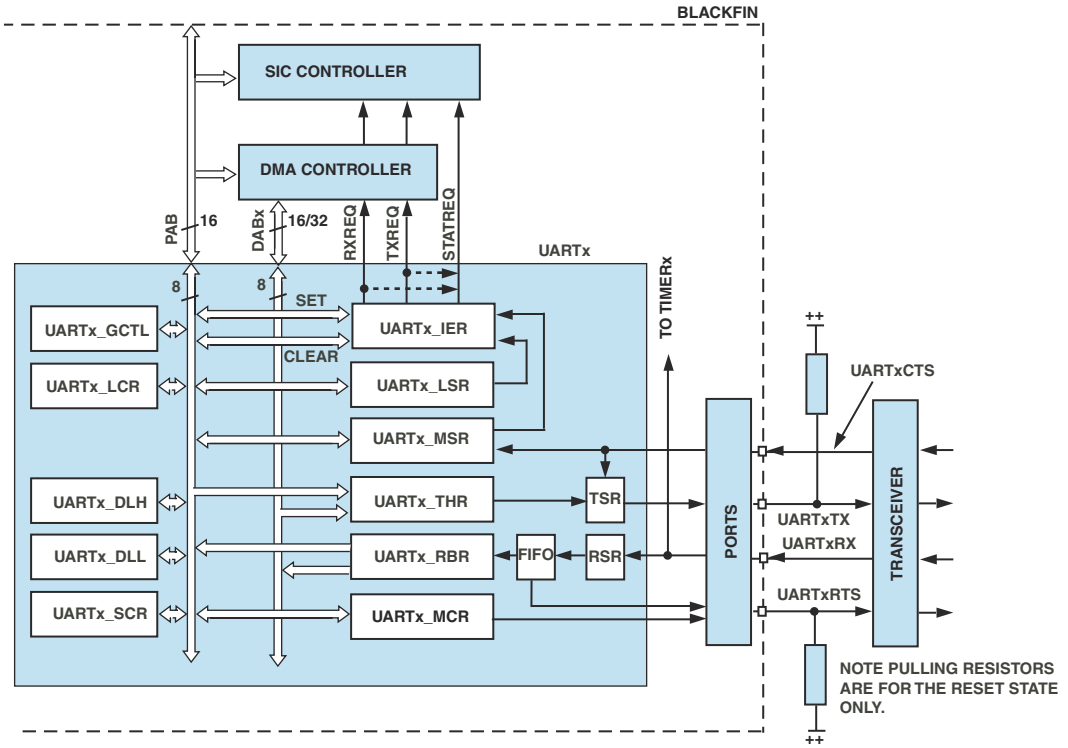


Figure 25-1. UART Block Diagram

External Interface

Each UART features an RX and a TX pin available through general-purpose ports. 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. Additionally, UART1 and UART3 feature a pair of UARTxCTS (clear to send, input) and UARTxRTS (request to send, output) signals for hardware flow control.

All UART signals are multiplexed and compete with other functions at pin level. Table 25-1 shows where the signal can be found and how they are enabled in the port control.

Table 25-1. UART Signals

Signal	Pin	Port Control	Autobaud Timer
UART0 TX	PE7	PORTE_MUX[15:14] = b#00 PORTE_FER[7] = 1	-
UART0 RX	PE8	PORTE_MUX[17:16] = b#00 PORTE_FER[8] = 1	Timer 0 (TACI0)
UART1 TX	PH0	PORTH_MUX[1:0] = b#00 PORTH_FER[0] = 1	-
UART1 RX	PH1	PORTH_MUX[3:2] = b#00 PORTH_FER[1] = 1	Timer 1 (TACI1)
UART1 RTS	PE9	PORTE_MUX[19:18] = b#00 PORTE_FER[9] = 1	-
UART1 CTS	PE10	PORTE_MUX[21:20] = b#00 PORTE_FER[10] = 1	-
UART2 TX	PB4	PORTB_MUX[9:8] = b#00 PORTB_FER[4] = 1	-
UART2 RX	PB5	PORTB_MUX[11:10] = b#00 PORTB_FER[5] = 1	Timer 2 (TACI2)
UART3 TX	PB6	PORTB_MUX[13:12] = b#00 PORTB_FER[6] = 1	-
UART3 RX	PB7	PORTB_MUX[15:14] = b#00 PORTB_FER[7] = 1	Timer 3 (TACI3)
UART3 RTS	PB2	PORTB_MUX[5:4] = b#00 PORTB_FER[2] = 1	-
UART3 CTS	PB3	PORTB_MUX[7:6] = b#00 PORTB_FER[3] = 1	-

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. For more information see Chapter 7, "Direct Memory Access".

All UART registers are 8 bits wide. They connect to the PAB bus. The UARTx_RBR and UARTx_THR registers also connect to one of the DABx busses. While UART0 and UART1 connect to the DAB16 bus, UART2 and UART3 connect to the DAB32 bus.

By default, no DMA channels are assigned to UART2 and UART3. To assign, program the PMAP crossbar in the DMAx_PERIPHERAL_MAP register of the desired DMA channels.

Each UART has three interrupt outputs. The transmit request and receive request outputs can function as DMA requests and connect to the DMA controller. Therefore, if the DMA is not enabled, the DMA controller simply forwards the request to the SIC controller. The status interrupt output connects directly to the SIC controller.

When no DMA channel is assigned, a UART has only one interrupt output. To modify, set the EGLSI bit in the UARTx_GCTL register to redirect transmit and receive requests to the status interrupt output.

Every UART's RX pin is also sensed by the alternative capture input (TACIx) of one of the general-purpose timers. Table 25-1 shows the assignment. In capture mode, the timers can be used to detect the bit rate of the received signal. See "Autobaud Detection" on page 25-20.

Description of Operation

The sections that follow 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 with the least significant bit (LSB) first.

Figure 25-2 shows a typical physical bitstream measured on one of the TX pins.

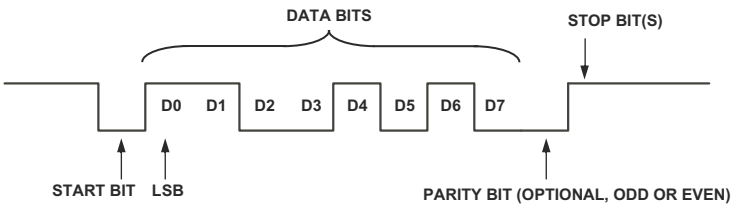


Figure 25-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

Aside from the standard UART functionality, the UART also supports serial data communication by way of infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

IrDA support is enabled by setting the IREN bit in the 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 characterized by the formula that follows with start, stop, and parity bits appended as defined by the UARTx_LCR register:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDB0)} \times Divisor}$$

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, 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. If no DMA channel

is assigned to the UART, the EGLSI bit in the UARTx_GCTL register can redirect the receive and transmit interrupts to the UART status interrupt alternatively.

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 all 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. An interrupt can be generated by that time either through the status interrupt channel when the ETFI bit is set, or through the DMA controller when enabled by the EDTPTI bit.

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.

The UART receiver is sensing the falling edges of the RX input. When an edge is detected, the receiver starts sampling the RX input according to the bit rate and the EDBO bit settings. The start bit is sampled close to its midpoint. If sampled low, a valid start condition is assumed. Otherwise, the detected falling edge is discarded.

After detection of the start bit, the received word is shifted into the internal shift register (RSR) at a bit rate characterized by the following formula:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDB0)} \times Divisor}$$

After the corresponding stop bit is received, the content of the RSR register is transferred through the 4-deep receive FIFO to the UARTx_RBR register, shown in Figure 25-13. 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 receive FIFOs and the UARTx_RBR registers can be seen as a five-stage receive buffer. If the stop bit of the 6th word is received before software reads the UARTx_RBR register, an overrun error is reported. The overrun case protects data in the UARTx_RBR and receive FIFO from being overwritten by further data until the OE bit is cleared by software. The data in the RSR register, however, is immediately destroyed as soon as the overrun occurs.

If enabled by the ERBFI bit in the UARTx_IER register, 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 no DMA channel is assigned to the UART, the EGLSI bit in the UARTx_GCTL register can redirect the receive and transmit interrupts to the UART status interrupt alternatively.

The state of the five-deep receiver buffer (including UARTx_RBR) can be monitored by the receiver FIFO count status (RFCS) bit in the UARTx_MSR register. The buffer's behavior is controlled by the receive FIFO interrupt threshold (RFIT) bit in the UARTx_MCR register. If RFIT is zero, the RFCS bit is set when the receive buffer holds two or more words. If RFIT is set, the RFCS bit is set when the receive buffer holds four or more words. The RFCS bit is cleared by hardware when core or DMA read the UARTx_RBR register and when the buffer is flushed below the level of two (RFIT=0) or four (RFIT=4). If the associated interrupt bit ERFCI is enabled, status interrupt is reported when the RFCS bit is set.

If errors are detected during reception, an interrupt can be requested to a the status interrupt output. This status interrupt request goes directly to the SIC interrupt controller. Status interrupt requests are enabled by the ELSI bit in the UARTx_IER_SET 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)

The sampling clock is 16 times faster than the bit clock. The receiver over samples every bit 16 times and does a majority decision based on the mid three samples. This improves immunity against noise and hazards on the line. Spurious pulses of less than two times the sampling clock period are disregarded.

Normally, every incoming bit is sampled at exactly the 7th, 8th and 9th sample clock. If, however, the EDBO bit is set to 1 to achieve better bit rate granularity and accuracy as required at high operation speeds, the bits are one roughly sampled at 7/16th, 8/16th and 9/16th of their period. Hardware design should ensure that the incoming signal is stable between 6/16th and 10/16th of the nominal bit period.

Reception is started when a falling edge is detected on the UARTxRX input pin. The receiver attempts to see a start bit. The data is shifted into the internal RSR register. After the 9th sample of the first stop bit is processed, the received data is copied to the 5-stage receive buffer and the RSR recovers for further data.

The receiver samples 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.

Hardware Flow Control

To prevent the UART transmitter from sending data while the receiving counterpart is not ready, a RTS/CTS hardware flow control mechanism is supported. The UARTxRTS (request to send) signal is an output that connects to the communication's partner UARTxCTS (clear to send) input. If data transfer is bidirectional, the handshake is as shown in Figure 25-3.

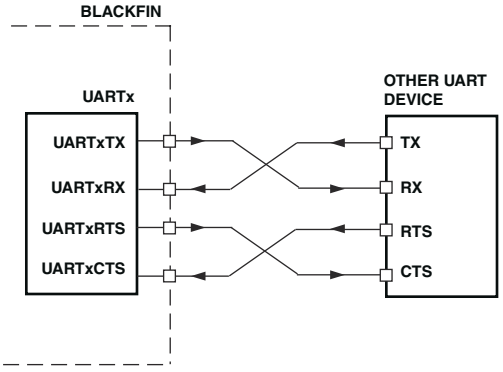


Figure 25-3. UART Hardware Flow

Regardless of whether working in DMA or non-DMA mode, the receiver can deassert the UARTxRTS signal to indicate that its receive buffer is getting full. Further data may cause an overrun error. Consequently, the transmitter pauses transmission when the UARTxCTS input is in deasserted state. On ADSP-BF54x processor processors, UART1 and UART3, if present, feature a pair of RTS/CTS pins each. Automatic hardware flow control can be enabled individually for receiver and transmitter by the UARTx_MCR register's ARTS and ACTS bits.

The signals are usually active low, that is, transmission is halted when the pin state is high. The polarity of the UARTxCTS and UARTxRTS pins can be inverted by setting the FCPOL bit in the UARTx_MCR register. If ACTS is enabled, the UARTxCTS bit in the UARTx_MSR register holds the complement value (FCPOL = 0) or the value (FCPOL = 1) of the UARTxCTS input pin. In either case the UARTxCTS bit reads 1 when the external device is ready to receive data. The delta CTS (DCTS) bit is a sticky version of the UARTxCTS bit that is set high when the UARTxCTS bit transitions from 0 to 1. It can request a status interrupt and is cleared by software with a W1C operation. If the TX handshaking protocol is enabled (bit ACTS = 1), the UART hardware pauses transmission if the UARTxCTS bit is zero. If the UARTx-CTS input is deasserted, the transmitter still completes transmission of the

data work currently held in the internal TSRx register, but does not continue with the data in UARTx_THR. If the UARTxCTS is asserted again, the transmitter resumes and loads the content of UARTx_THR into TSRx.

If the RX handshaking protocol is enabled (bit ARTS = 1 in the UARTx_MCR register), the UARTxRTS output pin is toggled automatically by the receiver's hardware. The pin's assertion and de-assertion timing is controlled by the receive FIFO RTS threshold (RFRT) bit in the UARTx_MCR register. If RFRT is cleared, the UARTxRTS pin is de-asserted when the receive buffer already holds two words and a third start bit is detected. The UARTxRTS pin is asserted again when the buffer does not contain any more data than the word in the UARTx_RBR register. If RFRT is set, the UARTxRTS pin is de-asserted when the receive buffer already holds four words and a fifth start bit is detected. The UARTxRTS is re-asserted when the buffer contains less than four words. Hardware guarantees minimal UARTxRTS de-assertion pulse width of at least the number of data bits as defined by the WLS bit field in the UARTx_LCR register.

If ACTS = 0, the TX handshaking protocol is disabled, and the UART transmits data as long as there is data to transmit, regardless of the value of UARTxCTS. With ACTS = 0 software can pause on-going transmission by setting the XOFF bit in the UARTx_MCR register.

If ARTS = 0, the UARTxRTS pin is not generated automatically by hardware. The UARTxRTS output can then still be manually controlled by the MRTS bit in the UARTx_MCR register.

On reset, when the UART is not yet enabled and the port multiplexing has not been programmed, the UARTxRTS pin is not driven. Some applications may require the UARTxRTS signal to be pulled to either state by a resistor during reset.

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 25-4. The final IrDA pulse is fed to the off-chip infrared driver.

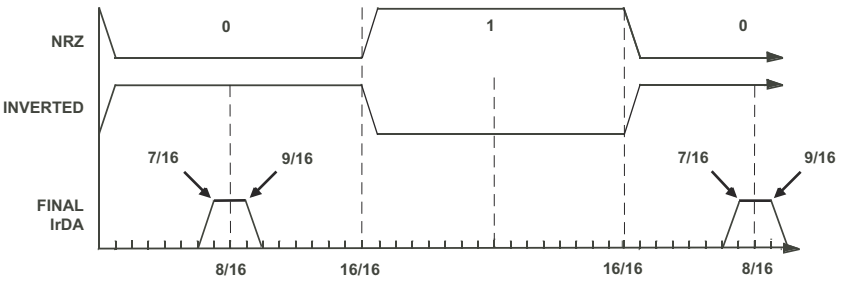


Figure 25-4. IrDA Transmit Pulse

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 25-2 on page 25-19, 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 25-5 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.

In the IrDA mode the EDBO bit is ignored. The sample frequency is always exactly 16 times the bit rate.

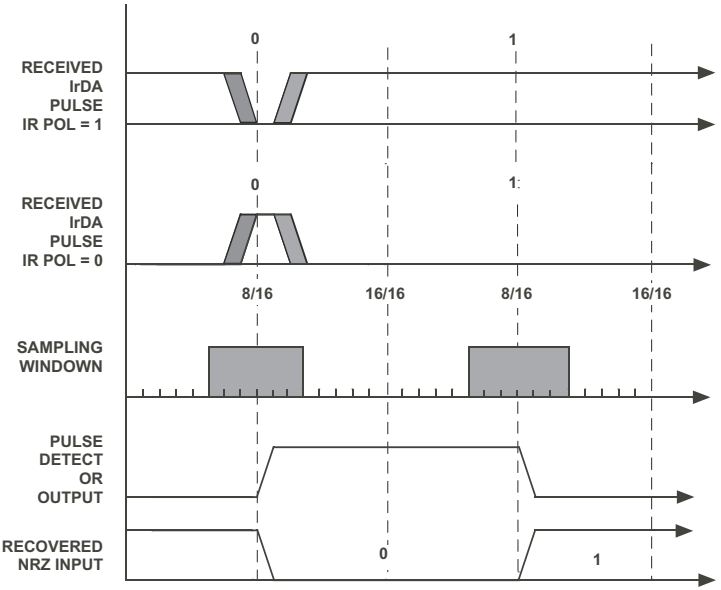


Figure 25-5. IrDA Receiver Pulse Detection

Interrupt Processing

Each UART module has three interrupt outputs. One is dedicated for transmission, one for reception, and the third is used to report status events. As shown in Figure 25-1 on page 25-3, the transmit and receive requests are routed through the DMA controller. The status request goes directly to the SIC controller.

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. For more information see Chapter 7, “Direct Memory Access”.

On ADSP-BF54x processor processors not all UARTs have a DMA channel assigned by default. Even if disabled, a DMA channel is still required to forward the DMA requests to the SIC controller as interrupt requests (see Figure 25-1 on page 25-3). Also, if no DMA channel is assigned, the UART loses its normal receive and transmit interrupt functionality.

To operate in interrupt mode without assigned DMA channels, set the EGLSI bit in the UARTx_GCTL register. This setup redirects receive and transmit requests to the status interrupt output. The status interrupt goes directly to the SIC controller without being routed through the DMA controller.

Transmit interrupts are enabled by the ETBEI bit in the UARTx_IER_SET register. If set, the transmit request is asserted along with the THRE bit in the UART_LSR, 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_SET 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 in the UARTx_IER_CLEAR register if the string transmission has completed. For more information, see “DMA Mode” on page 25-24.

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 clearing the ETBEI bit in the UARTx_IER_CLEAR register.

Receive interrupts are enabled by the ERBFI bit in the UARTx_IER_SET register. If set, the receive request is asserted along with the DR bit in the UART_LSR register, indicating that new data is available in the UARTx_RBR register. When software reads the UARTx_RBR, hardware clears the DR bit again which in turn clears the receive interrupt request.

The UART status interrupt channels are used for multiple purposes:

- Line Status Interrupts
- Flow Control Interrupts
- Receive FIFO Threshold Interrupts
- Transmission Finished Interrupt

Line status interrupts are enabled by the ELSI bit in the UARTx_IER_SET register. If set, the status interrupt request is asserted with any of the BI, FE, PE or OE receive errors bits in the UART_LSR register. Refer to "Line Status (UARTx_LSR) Registers" on page 25-34 for details. The error bits in the UARTx_LSR register are cleared by W1C operation. Once all error conditions are cleared the interrupt request de-asserts.

The receive FIFO count interrupt is enabled by the ERFCI bit in the UARTx_IER_SET register. If set, a status interrupt is generated when the RFCS is active. The RFCS bit indicates a receive buffer threshold level. If the RFIT bit in the UARTx_MCR register is cleared, software can safely read two words out of the UARTx_RBR register by the time the RFCS interrupt occurs. If the RFIT bit is set, software can safely read four words. The interrupt and the RFCS bit clear when the UARTx_RBR is read sufficient times, so that the receive buffer drains below the threshold of two (RFIT = 0) or four (RFIT = 1). Because in DMA mode a status service routine may not be permitted to read UARTx_RBR, this interrupt is only recommended in non-DMA mode. In DMA mode, use this functionality for error recovery only.

The UARTxCTS interrupts are enabled by the EDSSI bit in the UARTx_IER_SET register. If active, a status interrupt is generated when the sticky SCTS bit in the UARTx_MSR register is set, indicating that the transmitter's UARTxCTS input been re-asserted. A W1C operation to the SCTS bit clears the interrupt request.

A transmission finished interrupt is enabled by the ETFI bit in the UARTx_IER_SET register. If active, a status interrupt request is asserted when the TFI bit in the UARTx_LSR register is set. TFI is the sticky version of the TEMT bit, indicating that a byte that started transmission has completely finished. The interrupt request is cleared by a W1C operation to the TFI bit.

Bit Rate Generation

The UART clock is enabled by the UCEN bit in the UARTx_GCTL register.

The sample clock is characterized by the system clock (SCLK) and the 16-bit divisor. The divisor is split into the 8-bit UARTx_DLL and the UARTx_DLH registers. These registers form a 16-bit divisor.

By default every serial bit is over sampled 16 times. The bit clock is 1/16th of the sample clock. If not in IrDA mode the bit clock can equal the sample clock if the EDBO bit in the UARTx_GCTL register is set, so that the following applies:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDB0)} \times Divisor}$$

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

Table 25-2 provides example divide factors required to support most standard baud rates.

Careful selection of SCLK frequencies, that is, even multiples of desired bit rates, can result in lower error percentages.

Setting the bit clock equal to the sample clock (EDBO = 1) improves bit rate granularity and enables the Blackfin bit clock to more closely match the bit rate of the communication partner. There is, however, a disadvantage—the power dissipation is higher. Also the sample points may not be that accurate. It is recommended to use EDBO = 1 mode only when bit rate accuracy is not acceptable in EDBO = 0 mode.

The EDBO = 1 mode is not intended to increase operation speed beyond the electrical limitations of the asynchronous UART transfer protocol.

Table 25-2. UART Bit Rate Examples With 133 MHz SCLK

Bit Rate	Dfactor = 16			Dfactor = 1		
	DL	Actual	% Error	DL	Actual	% Error
2400	3464	2399.68	0.013	55417	2399.99	0.001
4800	1732	4799.36	0.013	27708	4800.06	0.001
9600	866	9598.73	0.013	13854	9600.12	0.001
19200	433	19197.46	0.013	6927	19200.23	0.001
38400	216	38483.80	0.218	3464	38394.92	0.013
57600	144	57725.69	0.218	2309	57600.69	0.001
115200	72	115451.39	0.218	1155	115151.52	0.042
921600	9	923611.11	0.218	144	923611.11	0.218
1500000	6	1385416.67	7.639	89	1494382.02	0.375
3000000	3	2770833.33	7.639	44	3022727.27	0.758
6250000	1	8312500.00	33.000	21	6333333.33	1.333

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 UARTxRX pin by an external device. For more information see Chapter 10, “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 UARTxRX pin by setting the LOOP_ENA 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 bit rate divider for the UART by using the following formula:

$$DIVISOR = \frac{\dot{TIMERx\_WIDTH}}{16^{(1-EDB0)} \times Number\ of\ captured\ UART\ bits}$$

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 25-6.

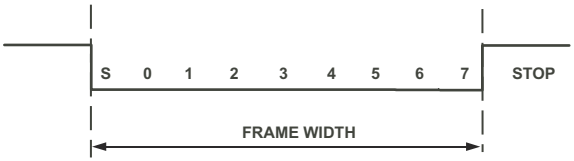


Figure 25-6. Autobaud Detection Character 0x00

Because the example frame in Figure 25-6 encloses 8 data bits and 1 start bit, apply the following formula:

$$DIVISOR = \frac{TIMERx\_WIDTH}{16^{(1-EDB0)} \times 9}$$

Real UARTx 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 25-7, 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 following:

- Divisor = TIMERx_PERIOD >> 7 if EDBO = 0
- Divisor = TIMERx_PERIOD >> 3 if EDBO = 1

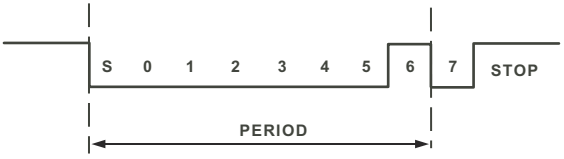


Figure 25-7. Autobaud Detection Character 0x40

An example is provided in Listing 25-2 on page 25-52.

Programming Model

The following sections describe the 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 a limited number of characters at a time.

To prevent any loss of data and misalignments of the serial data stream, 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, an overrun condition protects the already received data from being overwritten by new data until the OE bit is cleared by software. Only the content of the RSR register can be overwritten in the overrun case.

The TEMT bit can be interrogated to see whether any transmission is ongoing. The TEMT bit's sticky counterpart TFI tells whether the transmit buffer has drained and can trigger a status interrupt, if required.

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. Since read operations from UARTx_LSR registers have no side effects, different software threads can interrogate these registers without mutual impacts. Polling the SIC_ISRx register without enabling the interrupts by SIC_MASKx 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, UARTx RX, and UART status. The independent interrupts can be enabled individually by the UARTx_IER_SET and UARTx_IER_CLEAR register pair. The UCEN bit must be set to enable UART transmit interrupts.

The ISRs can evaluate the status bits in the UARTx_LSR and UARTx_MSR registers to determine the signalling interrupt source. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 25-15 on page 25-43.

To reduce interrupt frequency on the receive side in non-DMA mode, the ERFCI status interrupt may be used as an alternative to the regular ERBFI receive interrupt. Hardware ensure that at least two (if RFIT = 0) or four (if RFIT = 1) words are available in the receive buffer by the time the interrupt is requested.

DMA Mode

In this mode, separate receive (UARTxRX) and transmit (UARTxTX) 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 the transmit and 9 words at the receive side 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 7, "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_SET 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 status 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 is drained to the UART module. The UART TX DMA interrupt service routine is allowed to disable the DMA or to clear the ETBEI control bit only when the SYNC bit is set, otherwise up to four data bytes might be lost.

When the ETBEI bit is set in the UARTx_IER_SET register, an initial transmit DMA request is issued immediately. It is common practice to clear the ETBEI bit by the DMA's service routine.

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 synchronization with the TEMT bit.

The ADSP-BF54x processor UART implementation provides new functionality to avoid expensive polling of the TEMT bit. The EDTPTI bit in the UARTx_IER_SET register enables the TEMT bit to trigger a DMA interrupt. To delay the DMA completion interrupt until the last data word of a STOP DMA has left the UART, keep the DMA's DI_EN bit cleared and set the EDTPTI bit instead. Then, the normal DMA completion interrupt is suppressed. Later, the TEMT event triggers a DMA interrupt after the DMA's last word has left the UART transmit buffers. If DI_EN and EDTPTI are set, when finishing STOP mode, the DMA requests two interrupts.

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. Normally, the UARTxTX DMA completion interrupt is generated after the last byte is copied from the memory into the DMA FIFO. The UARTxTX DMA interrupt service routine is not yet permitted to disable the DMA enable bit DMAEN. 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 UARTxTX 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. Using the EDTPTI bit can avoid expensive status bit polling, alternatively.

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 25-3 provides an overview of the UART registers.

Unlike on ADSP-BF53x processors, register addresses are not shared on ADSP-BF54x processor processors. Each register has its own MMR address. Consequently, the DLAB bit is not present on ADSP-BF54x processor processors' UARTx_LSR registers. Software must use 16-bit word load/store instructions to access these registers.

Furthermore, the interrupt processing differs from ADSP-BF53x processors. Error bits in status registers do not clear on register reads implicitly, rather they are cleared by write-1-to-clear (W1C) operations. The UARTx_IIR register is not present at all. The interrupt enable register has separate set and clear ports, so that separate receive, transmit, and status interrupt service routines can enable or set masks individually.

Transmit and receive channels are both buffered. The UARTx_THR registers buffer the transmit shift registers (TSR). The UARTx_RBR registers and an additional 4-stage receive FIFO buffer the receive shift register (RSR). The shift registers are not directly accessible by software.

Table 25-3. ADSP-BF54x vs. ADSP-BF53x UART Register

Register Name	Address Offset		Description
	ADSP-BF54x	ADSP-BF53x	
UARTx_DLL	0x00	0x00 DLAB=1	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
UARTx_DLH	0x04	0x00 DLAB=1	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46

Table 25-3. ADSP-BF54x vs. ADSP-BF53x UART Register (Cont'd)

Register Name	Address Offset		Description
	ADSP-BF54x	ADSP-BF53x	
UARTx_GCTL	0x08	0x24	"Global Control (UARTx_GCTL) Registers" on page 25-50
UARTx_LCR	0x0C	0x0C	"Line Control (UARTx_LCR) Registers" on page 25-29
UARTx_MCR	0x10	0x10	"Modem Control (UARTx_MCR) Registers" on page 25-32
UARTx_LSR	0x14	0x14	"Line Status (UARTx_LSR) Registers" on page 25-34
UARTx_MSR	0x18	N/A	"Modem Status (UARTx_MSR) Registers" on page 25-37
UARTx_SCR	0x1C	0x1C	"UART Scratch (UARTx_SCR) Registers" on page 25-49
UARTx_IER_SET	0x20	N/A	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
UARTx_IER_CLEAR	0x24	N/A	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
UARTx_IER	N/A	0x04 DLAB=0	"Line Control (UARTx_LCR) Registers" on page 25-29
UARTx_THR	0x28	0x00 DLAB=0	"Transmit Hold (UARTx_THR) Registers" on page 25-39
UARTx_RBR	0x2C	0x00 DLAB=0	"Receive Buffer (UARTx_RBR) Registers" on page 25-40
UARTx_IIR	N/A	0x08	"Line Control (UARTx_LCR) Registers" on page 25-29

Line Control (UARTx_LCR) Registers

The line control (UARTx_LCR) registers, shown in Figure 25-8, control the format of received and transmitted character frames.

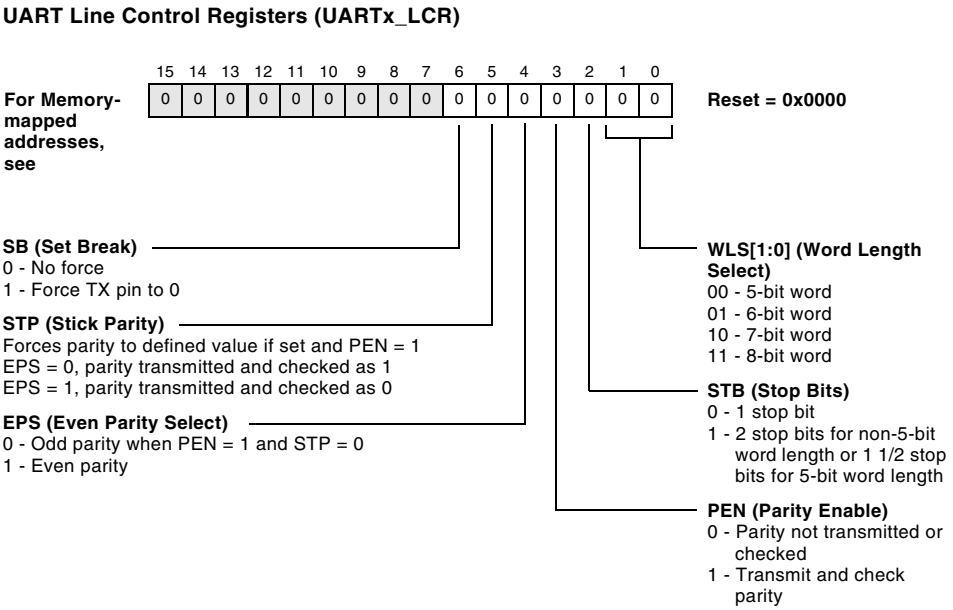


Figure 25-8. UART Line Control Registers

Table 25-4. UART Line Control Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
UART0_LCR	0xFFC0 040C
UART1_LCR	0xFFC0 200C
UART2_LCR	0xFFC0 210C
UART3_LCR	0xFFC0 310C

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 25-5 summarizes polarity behavior assuming 8-bit data words (WLS = 3).

Table 25-5. UART Parity

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

Table 25-5. UART Parity (Cont'd)

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
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 UARTxTX pin to low asynchronously, regardless of whether or not data is currently transmitted. It functions even when the UART clock is disabled. Since the UARTxTX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

Modem Control (UARTx_MCR) Registers

The modem control (UARTx_MCR) registers control the UART port, as shown in Figure 25-9. Partial modem functionality is supported to allow for hardware flow control and loopback mode.

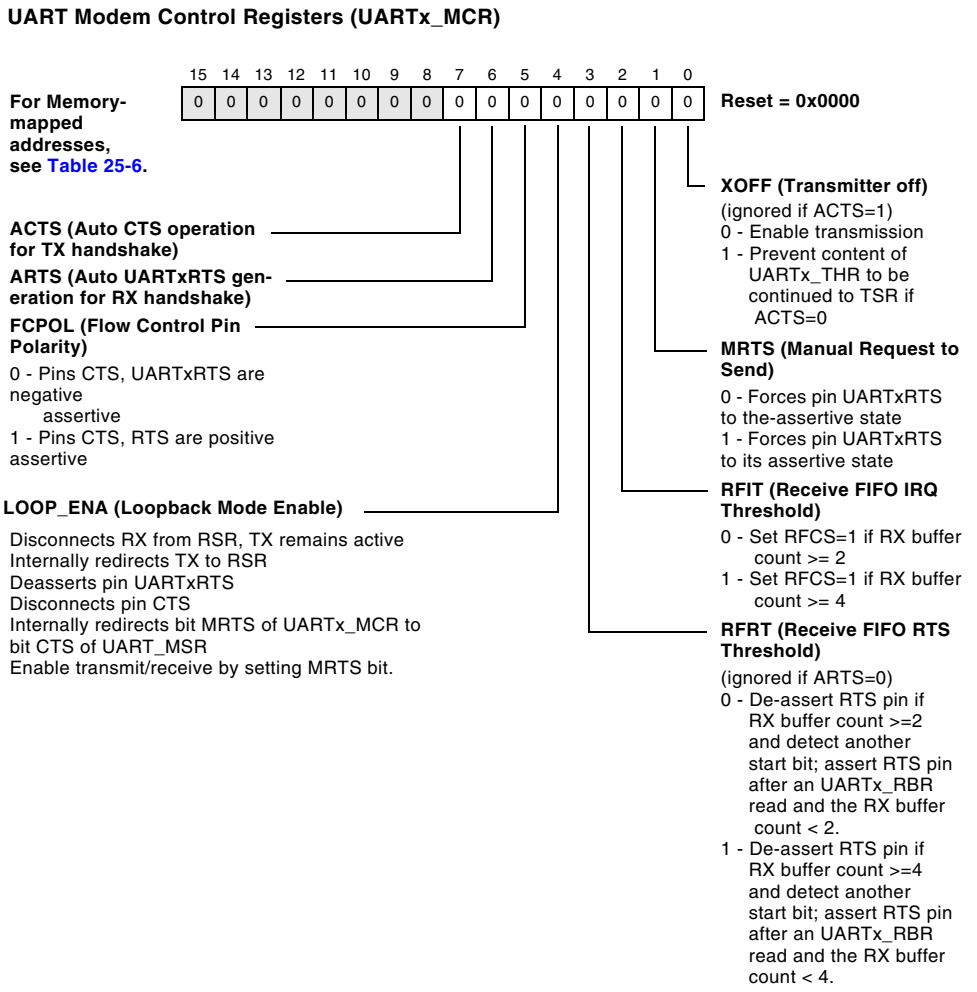


Figure 25-9. UART Modem Control Registers

Table 25-6. UART Modem Control Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_MCR	0xFFC0 0410
UART1_MCR	0xFFC0 2010
UART2_MCR	0xFFC0 2110
UART3_MCR	0xFFC0 3110

The receive FIFO interrupt threshold (RFIT) bit controls the timing of the RFCS status bit. If RFIT = 0, the receive threshold is two. If RFIT = 1, the threshold is four words in the receive buffer.

The manual request to send (MRTS) bit controls the state of the UARTxRTS output pin only if ARTS = 0. A value of MRTS = 0 forces the UARTxRTS pin to its de-assertive state, signaling to the external device that the UART is not ready to receive. A value of MRTS = 1 forces the UARTxRTS pin to its assertive state, signaling to the external device that the UART is ready to receive.

The automatic RTS (ARTS) bit enables the receive buffer to control the RTS output depending on the threshold programmed by the RFTR bit. If RFRT = 0, the RTS signal is de-asserted when already two words are held by the receive buffer and a third start bit is detected. It is re-asserted if the buffer contains less than two words. If RFRT = 1, the RTS signal is de-asserted when already four words are held by the receive buffer and a fifth start bit is detected. The RTS signal is re-asserted if the buffer contains less than four words.

Similarly, the automatic CTS (ACTS) bit must be set to enable the CTS input pin for UARTxTX handshaking. If enabled, the CTS status bit in the UARTx_MSR register holds the value (if FCPOL = 1) or complement value (if FCPOL = 0) of the CTS input pin. The CTS status bit can be used to determine if the external device is ready to receive data (CTS = 1) or if it is busy (CTS = 0). If ACTS = 0, the UARTxTX handshaking protocol is disabled, and the UARTxTX line transmits data whenever there is data to send, regardless of the value of CTS. The transmitter off (XOFF) bit can be used to

pause an on-going transmission by software when ACTS = 0. Similarly to automatic CTS mode, the XOFF bit prevents the data in the UARTx_THR register from being continued to the TSR shift register. When ACS = 1, the XOFF bit is ignored. When ACTS = 0, the state of the CTS input signal is ignored.

The polarities of the UARTxCTS and UARTxRTS pins can be programmed using the FCPOL bit. If FCPOL = 0, the pins are negative asserted. If FCPOL = 1, the pins are positive asserted.

Loopback mode (LOOP_ENA = 1) disconnects the receiver's input from the UARTxRX pin, and internally redirects the transmit output to the receiver. The UARTxTX pin remains active and continues to transmit data externally as well. Loopback mode also forces the UARTxRTS pin to its de-assertive state, disconnects the UARTxCTS bit from the UARTxCTS input pin, and directly connects bit MRTS to bit UARTxCTS of the modem status register (UARTx_MSR). In loopback mode, writing a 1 to the MRTS bit sets bit UARTxCTS, DCTS and enable the UART's transmitter. Writing a 0 to the MRTS bit clears bit UARTxCTS and disable the UART's transmitter.

Line Status (UARTx_LSR) Registers

The line status (UARTx_LSR) registers contain UART status information as shown in Figure 25-10. Unlike the industrial standard, the ADSP-BF54x processor processor's UARTx_LSR register is not read only. Writes to this register can perform write-one-to-clear (W1C) operations on most status bits. Reading this register has no side effects.

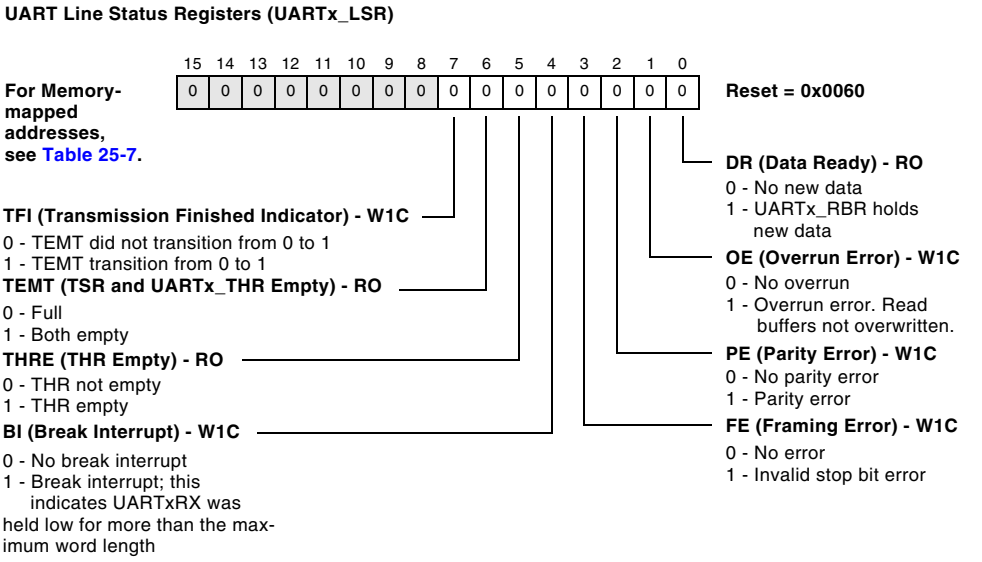


Figure 25-10. UART Line Status Registers

Table 25-7. UART Line Status Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_LSR	0xFFC0 0414
UART1_LSR	0xFFC0 2014
UART2_LSR	0xFFC0 2114
UART3_LSR	0xFFC0 3114

The DR (data ready) 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 (overrun error) bit indicates that further data is received while the internal receive buffer was full. It is set when sampling the stop bit of the 6th data word. To avoid overruns, read the UARTx_RBR register in time. In

DMA receive mode overruns are very unlikely to happen ever. Once an overrun occurs, the UARTx_RBR and receive FIFO are protected from being overwritten by new data until the OE bit is cleared by software. The content of receive shift register RSR, however, is lost as soon as the overrun occurs. The OE bit is sticky and can be cleared by W1C operations.

The PE (parity error) bit indicates that the received parity bit does not match the expected value. The PE bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations. Invalid parity bits can be simulated by setting the FPE bit in the UARTx_GCTL register.

The FE (framing error) bit indicates that the first stop bit is sampled. The FE bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations. Invalid stop bits can be simulated by setting the FFE bit in the UARTx_GCTL register.

The BI (break indicator) bit indicates that the first stop bit is sampled low and the entire data word, including parity bit, consists of low bits only. The BI bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations.

The THRE (transmit hold register empty) 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 (transmitter empty) 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 is completed. At that time it is safe to disable the UCEN bit or to three-state the off-chip line driver.

The TFI (transmission finished indicator) bit is a sticky version of the TEMT bit. While TEMT is automatically cleared by hardware when new data is written to the UARTx_THR register, the sticky TFI bit remains set until it is cleared by software (W1C). The TFI bit enables more flexible transmit interrupt timing.

Modem Status (UARTx_MSR) Registers

The modem status (UARTx_MSR) registers, shown in Figure 25-12, contains current states of the UART's external UARTxCTS pin and current status of the UART's internal receive buffers.

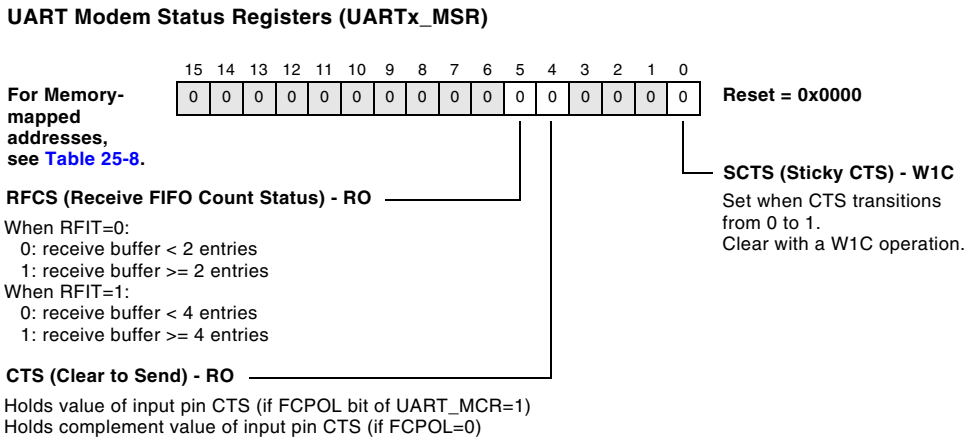


Figure 25-11. UART Modem Status Registers

Table 25-8. UART Modem Status Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_MSR	0xFFC0 0418
UART1_MSR	0xFFC0 2018
UART2_MSR	0xFFC0 2118
UART3_MSR	0xFFC0 3118

The UARTxCTS bit holds the value (if FCPOL = 1) or the complement value (if FCPOL = 0) of the UARTxCTS input pin. The ACTS bit in the UARTx_MCR register must be set to enable this feature. The core can read the value of UARTxCTS to determine if the external device is ready to receive (UARTxCTS = 1) or if it is busy (UARTxCTS = 0). If ACTS = 0, the UARTxTX handshaking protocol is disabled, and the UART transmits data as long as there is data to transmit, regardless of the value of UARTxCTS. When ACTS = 0, the software can pause transmission temporarily by setting the XOFF bit.

The SCTS bit is a sticky bit that is set high when UARTxCTS transitions from 0 to 1, and is cleared by software with a W1C operation. The SCTS bit can trigger a line status interrupt if enabled by the EDSSI bit in the UARTx_IER_SET register.

The receiver FIFO count status (RFCS) bit is set when the receive buffer holds more or equal entries than a certain threshold. The threshold is controlled by the RFIT bit in the UARTx_MCR register. If RFIT = 0, the threshold is two entries. If RFIT = 1, the threshold is four entries. The RFCS bit cleared when the UARTx_RBR register is read sufficient times until the buffer is drained below the threshold. The RFCS bit can trigger a status interrupt if enabled by the ERFCI bit in the UARTx_IER_SET register.

In loopback mode (LOOP_ENA = 1), the UARTxCTS bit is disconnected from the UARTxCTS input pin. Instead, it is directly connected to the MRTS bit of the UARTx_MCR register.

Previous implementations of the UART did not have this register. It is implemented to allow for hardware flow control between the UART and an external device.

Transmit Hold (UARTx_THR) Registers

The write-only transmit hold (UARTx_THR) registers, shown in Figure 25-12, is the UART's transmit buffer. The THRE bit in the UARTx_LSR registers indicate whether UARTx_THR is ready for new data. Writes to UARTx_THR automatically propagate to the internal TSR register as soon as TSR is ready. Then transmit operation is initiated immediately.

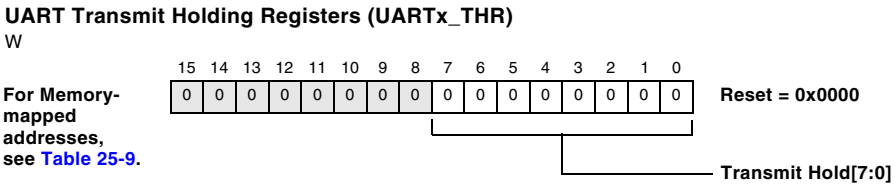


Figure 25-12. UART Transmit Holding Registers

Table 25-9. UART Transmit Holding Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_THR	0xFFC0 0428
UART1_THR	0xFFC0 2028
UART2_THR	0xFFC0 2128
UART3_THR	0xFFC0 3128

Receive Buffer (UARTx_RBR) Registers

The read-only UARTx_RBR registers, shown in Figure 25-13, is the UART's receive buffer. It is updated by the internal RSR register when a complete data word is received or when there is pending data in the receive FIFO. Newly available data is signalled by the DR bit in the UARTx_LSR register.

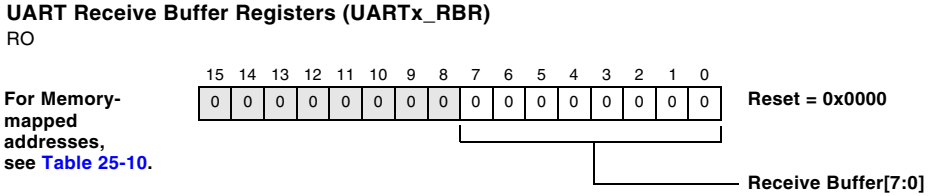


Figure 25-13. UART Receive Buffer Registers

Table 25-10. UART Receive Buffer Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_RBR	0xFFC0 042C
UART1_RBR	0xFFC0 202C
UART2_RBR	0xFFC0 212C
UART3_RBR	0xFFC0 312C

Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers

The interrupt enable register is not implemented as a data register. Instead it is controlled by the UARTx_IER_SET and UARTx_IER_CLEAR register pair. Writing ones to UARTx_IER_SET enables interrupts, writing UARTx_IER_CLEAR disables them. Reads from either register return the enabled bits. This way, different interrupt service routines can control transmit, receive, and status interrupts independently and gracefully.

The UARTx_IER registers, shown in Figure 25-14 and Figure 25-15, 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.

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. If no DMA channels are assigned to the UART, set the EGLSI bit in the UARTx_GCTL register to reroute transmit and receive interrupts to the status interrupt output.

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.

UART Interrupt Enable Set Registers (UARTx_IER_SET)

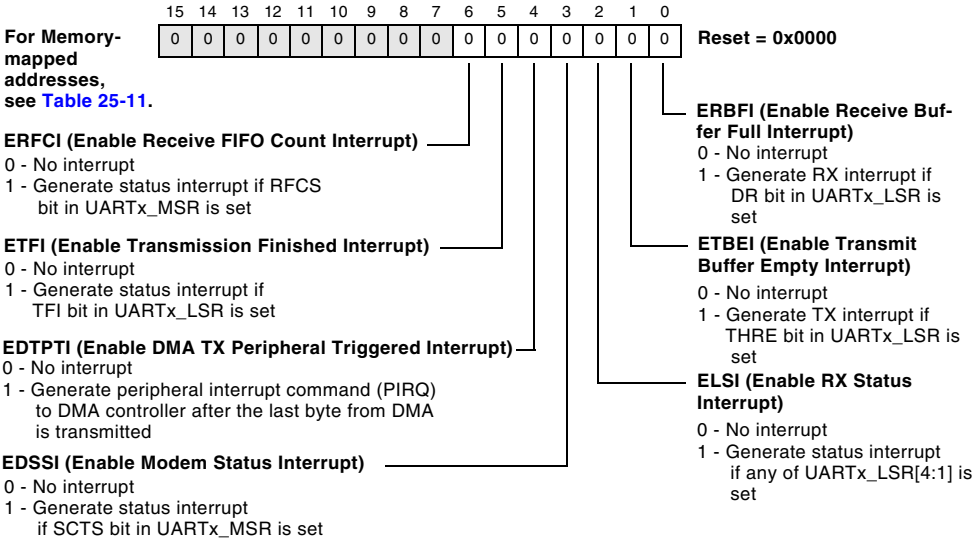


Figure 25-14. UART Interrupt Enable Set Registers

Table 25-11. UART Interrupt Enable Set Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_IER_SET	0xFFC0 0420
UART1_IER_SET	0xFFC0 2020
UART2_IER_SET	0xFFC0 2120
UART3_IER_SET	0xFFC0 3120

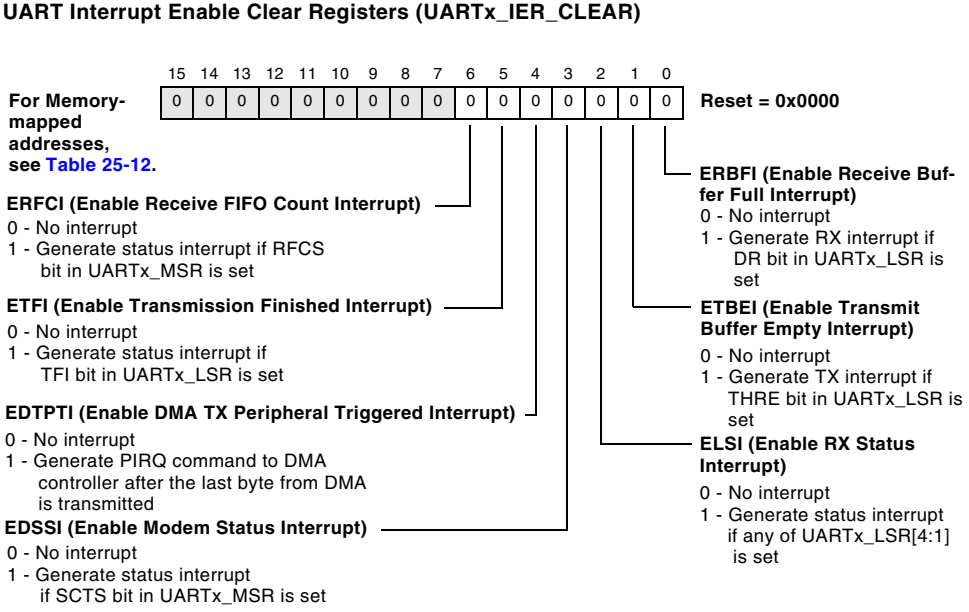


Figure 25-15. UART Interrupt Enable Clear Registers

Table 25-12. UART Interrupt Enable Clear Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_IER_CLEAR	0xFFC0 0424
UART1_IER_CLEAR	0xFFC0 2024
UART2_IER_CLEAR	0xFFC0 2124
UART3_IER_CLEAR	0xFFC0 3124

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)

The EDSSI bit enables a modem status interrupt on the same status interrupt channel when the SCTS bit in the UARTx_MSR register is set. This indicates CTS re-assertion. Write-1-to-clear (W1C) the SCTS bit to clear the interrupt request.

The ERFCI bit enables the receive buffer threshold interrupt if signalled by the RFCS bit. Read the UARTx_RBR register sufficient times to clear the interrupt request.

The ETFI bit enables interrupt generation on the status interrupt channel when both the transmit buffer register and transmit shift register are empty as indicated by the TFI bit in the UARTx_LSR register. The ETFI interrupt can be used to avoid expensive polling of the TEMT bit, when the UART clock or line drivers should be disabled after transmission has completed. W1C the TFI bit to clear the interrupt request. In DMA operation, the ETDPTI bit's functionality might be preferred.

The ETDPTI bit is required for DMA transmit operation only. It enables the DMA completion interrupt to be delayed until the data has left the UART completely. If set, it can generate a DMA interrupt by the time the TEMT bit goes high after the last DMA data word is transmitted.

If the ETDPTI bit is cleared, the DMA completion interrupt is generated when either the last data word is transferred from memory to the DMA FIFO (DMA's SYNC bit cleared) or when the last word has left the DMA FIFO (SYNC bit set). If ETDPTI is set, usually the DMA's DI_EN is not set in a STOP mode DMA. Thus, the normal completion interrupt is suppressed. Rather, the TEMT event is signalled through the DMA controller and triggers the DMA interrupt. If both, DI_EN and ETDPTI are set, two interrupts are requested at the end of a STOP mode DMA.

The UARTx_IIR registers are not present on this implementation. Signalling interrupt sources can be identified by interrogating UARTx_LSR and UARTx_MSR status registers.

Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers

The two 8-bit clock divisor latch registers (UARTx_DLH and UARTx_DLL) build a 16-bit clock divisor value. They divide the system clock SCLK down to the bit clock. These registers are shown in Figure 25-16.

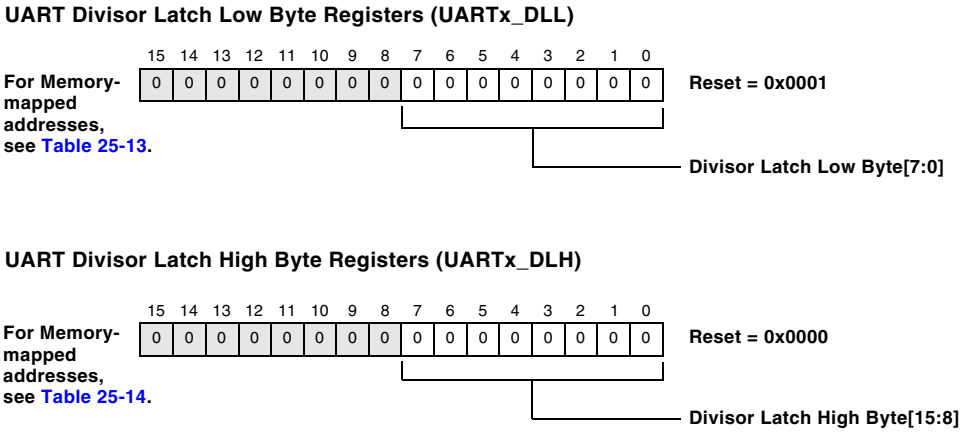


Figure 25-16. UART Divisor Latch Registers

Table 25-13. UART Divisor Latch Low Byte Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_DLL	0xFFC0 0400
UART1_DLL	0xFFC0 2000
UART2_DLL	0xFFC0 2100
UART3_DLL	0xFFC0 3100

Table 25-14. UART Divisor Latch High Byte Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_DLH	0xFFC0 0404
UART1_DLH	0xFFC0 2004
UART2_DLH	0xFFC0 2104
UART3_DLH	0xFFC0 3104

Note the 16-bit divisor formed by UARTx_DLH and UARTx_DLL resets to 0x0001, resulting in high clock frequency by default. If the UART is not used, disabling the UART clock saves power.

Note that the bit rate depends also on the EDBO bit in the UARTx_GCTL register. Refer to "Bit Rate Generation" on page 25-18.

UART Scratch (UARTx_SCR) Registers

The contents of the 8-bit scratch (UARTx_SCR) registers, shown in Figure 25-17, are reset to 0x00. They are used for general-purpose data storage and do not control the UART hardware in any way.

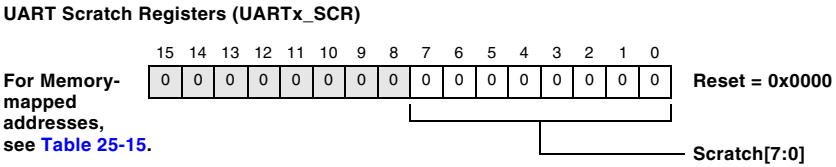


Figure 25-17. UART Scratch Registers

Table 25-15. UART Scratch Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_SCR	0xFFC0 041C
UART1_SCR	0xFFC0 201C
UART2_SCR	0xFFC0 211C
UART3_SCR	0xFFC0 311C

Global Control (UARTx_GCTL) Registers

The global control (UARTx_GCTL) registers, shown in Figure 25-18, contain the enable bit for internal UART clocks and for the IrDA mode of operation of the UARTs.

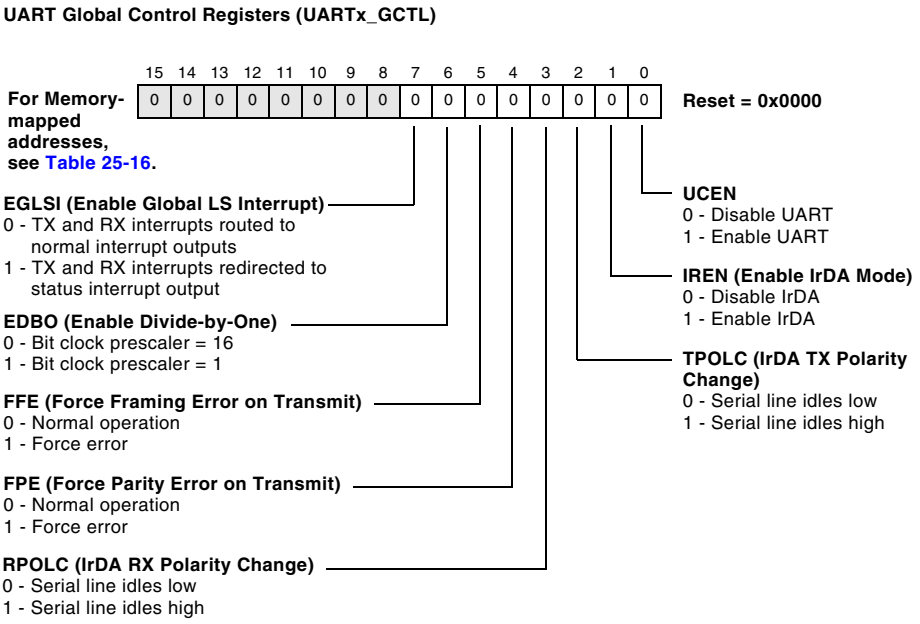


Figure 25-18. UART Global Control Registers

Table 25-16. UART Global Control Register Memory-mapped Addresses

Register Name	Memory-mapped Address
UART0_GCTL	0xFFC0 0408
UART1_GCTL	0xFFC0 2008
UART2_GCTL	0xFFC0 2108
UART3_GCTL	0xFFC0 3108

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 is 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.

The EDBO bit enables bypassing of the divide-by-16 prescaler in bit clock generation. This improves bit rate granularity, especially at high bit rates. See "Bit Rate Generation" on page 25-18. Do not set this bit in IrDA mode.

The EGLSI bit redirects TX and RX interrupt requests to the status interrupt output of the UART by ORing them with all other kinds of UART status interrupt requests. Set this bit when no DMA channel is associated with the UART. Enabling EGLSI disables the RX/TX interrupt channels and negates the EDTPTI bit.

Programming Examples

The following programming examples show how to use the UART.

The subroutine in Listing 25-1 shows a typical UART initialization sequence.

Listing 25-1. 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;

     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);              /* config 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 25-2 performs autobaud detection similarly to UART boot.

Listing 25-2. 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 25-3 performs autobaud detection using UART0 and TIMER0.

Listing 25-3. UART Autobaud Detection Parent Routine

```
   p0.l = lo(PORTE_FER); /* function enable on UART0 pins PE7 and
PE8 and PF1 */
    p0.h = hi(PORTE_FER); /* by default PORTE_MUX register is all
set */
    r0 = PE8 | PE7 (z)
    w[p0] = r0;
    p0.l = lo(UART0_GCTL); /* select UART 0 */
```

```
    p0.h = hi(UART0_GCTL);
    p1.l = lo(TIMER0_CONFIG); /* select TIMER 0 */
    p1.h = hi(TIMER0_CONFIG);
    call uart_autobaud;
    r0 >>= 7;       /* divide PERIOD value by (16 x 8) */
    call uart_init;
    ...
```

The subroutine in Listing 25-4 transmits a character by polling operation.

Listing 25-4. 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 25-5 to transmit a C-style string that is terminated by a null character.

Listing 25-5. 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 does not cause side effects on receive status bits as on former implementations.

In non-DMA interrupt operation, the three UART interrupt request lines may or may not be ORed together in the SIC controller or by the EGLSI control bit. If they had three different service routines, they may look as shown in Listing 25-6.

Listing 25-6. 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 = ETBEI (z) ;
    w[p0+UART0_IER_CLR] = r7; /* clear TX interrupt enable */
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:

isr_uart_error:
```

```
    [--sp] = astat;
    [--sp] = (r7:6);
    r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
    r6 = OE | BI | FE | PE (z);
    w[p0+UART0_LSR-UART0_GCTL] = r6;
    /* do something with the error */
    (r7:6) = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_error.end:
```

Listing 25-7 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 25-7. 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_IMASK0);
    p1.h = hi(SIC_IMASK0);       /* unmask interrupt in SIC */
    r0.l = 0x8000;
    r0.h = 0x0000;
```

```
    [p1] = r0;
    [--sp] = reti;        /* enable nesting of interrupts */

    p5.l = lo(DMA7_CONFIG);     /* setup DMA in STOP mode */
    p5.h = hi(DMA7_CONFIG);
    r7.l = lo(sHello);
    r7.h = hi(sHello);
    [p5+DMA7_START_ADDR-DMA7_CONFIG] = r7;
    r7 = length(sHello) (z);
    r7+= -1;    /* don't send trailing null character */
    w[p5+DMA7_X_COUNT-DMA7_CONFIG] = r7;
    r7 = 1;
    w[p5+DMA7_X_MODIFY-DMA7_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+DMA7_IRQ_STATUS-DMA7_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+DMA7_IRQ_STATUS-DMA7_CONFIG] = r7;
        r7 = ETBEI (z);
        w[p0+UART0_IER_CLEAR-UART0_GCTL] = r7;
        ssync;
        r7 = [sp++];
        astat = [sp++];
        rti;
    isr_uart_tx.end:
```

26 USB OTG CONTROLLER

This chapter describes the 6-pin USB OTG interface for the USB OTG controller.

This chapter includes the following sections:

- "Overview" on page 26-1
- "Interface Overview" on page 26-3
- "Description of Operation" on page 26-12
- "Functional Description" on page 26-54
- "Programming Model" on page 26-56
- "USB OTG Registers" on page 26-97
- "References" on page 26-148
- "Glossary of USB Terms " on page 26-148

Overview

The USB OTG controller provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data using a point-to-point USB connection without the need for a personal computer host.

The USB controller can operate in a traditional USB peripheral-only mode as well as the host mode presented in the On-The-Go (OTG) supplement[1] to the USB 2.0 Specification[2]. In host mode, the USB module supports transfers at high-speed (480Mbps), full-speed (12Mbps), and low-speed (1.5Mbps) rates. Peripheral mode supports the high- and full-speed transfer rates.

The USB controller uses a peripheral bus slave interface to access its control and status registers as well as read and write to the endpoint packet buffers. Data is transferred to and from the USB controller through any of the seven transmit and seven receive endpoint FIFOs, EP1 – EP7, providing a total of 14 data endpoints. A DCB/DEB bus master interface provides eight DMA channels to provide a more efficient means of transferring large amounts of data between the controller and the Blackfin processor's memory map.

Features

The USB controller provides the following features:

- low speed, full speed, high speed rates supported
- one bidirectional control endpoint
- seven transmit and seven receive unidirectional endpoints
- 7.232K Bytes of FIFOs for packet buffering
- eight DMA master channels
- three top-level maskable general purpose interrupts
- one asynchronous wakeup interrupt

---

[1] On-The-Go Supplement to the USB 2.0 Specification, Rev 1.0a; June 24, 2003; USB-IF

[2] Universal Serial Bus Specification 2.0

- VBUS control interrupts for external analog VBUS control
- software-controlled clock control on each endpoint for power reduction
- session request protocol (SRP) and host negotiation protocol (HNP) capability
- host transaction scheduling in hardware
- soft connect/disconnect feature
- full- and high-speed physical layer UTMI+ level 2 interface for on-chip PHY
- backwards compatible with existing USB 1.1 hosts

The number of active endpoints at one time is only limited by device requirements or system bandwidth, because each endpoint operates independently from the next. The maximum buffer size per endpoint is 1024 bytes. Software determines the type of transfer for each endpoint individually and also the manner in which it is transferred between the USB controller and memory (DMA or interrupt-based). Endpoint zero is used solely for receive and transmit control transfers, which are used for device configuration and information gathering.

Interface Overview

The USB controller operates in either of two USB operation modes (peripheral or host mode) at a given time.

In peripheral mode, the USB controller encodes, decodes, checks, and directs all USB packets sent and received, responding appropriately to host requests. Data is transferred from the processor core memory into the device's TX FIFOs to be transmitted onto USB as IN packets. In the other direction USB OUT packets are received into the RX FIFOs (having been

sent from the host) and transferred to system memory for processing or storage. In peripheral mode, the USB controller acts as a slave device to another USB host; either a personal computer or another OTG host controller.

When operating in host mode, the USB controller uses simple hosting capabilities to master point-to-point connections with another USB peripheral, initiating transfers on the bus for the peripheral to respond. USB IN packets are received into the RX FIFOs to be moved into the processor core memory, and data written into TX FIFOs is transmitted onto the bus as USB OUT packets. In this mode, the USB controller encodes, decodes, and checks USB packets sent and received. The controller automatically schedules isochronous and interrupt transfers from the endpoint buffers such that one transaction is performed every *n* frames, where *n* represents the polling interval programmed for the endpoint.

Figure 26-1 shows the main functional blocks within the USB controller and its interfaces to the processor core, USB controller RAM, and USB OTG PHY.

Any of the endpoints can be programmed to be written to or read from using the DMA master channels to provide the most efficient means of transferring data between the controller and on-chip memory. USB endpoints 0 through 7 have DMA interrupt lines (`USB_DMAxINT`) providing a total of eight DMA request lines. Three top-level maskable interrupts are provided, each of which can be sourced from any or all of transmit endpoint status, receive endpoint status or global USB status. Details of these can be found in "Interrupts" on page 26-8.

The USB controller uses the peripheral bus to access control and status registers and FIFOs from a slave perspective and to transfer data between the USB engine and on-chip memory as a master. The MMR peripheral data bus is 16-bits wide, the DMA DCB/DEB data bus is also 16-bits wide. Using the 16-bit wide data bus, the USB controller to processor core interface translates into either half word transfers (for both CSR and FIFO addresses) or byte transfers (FIFO addresses only).

The USB controller's RAM interface supports a single block of synchronous single-port RAM used to buffer the USB packets. 7.232K bytes of SRAM are available.

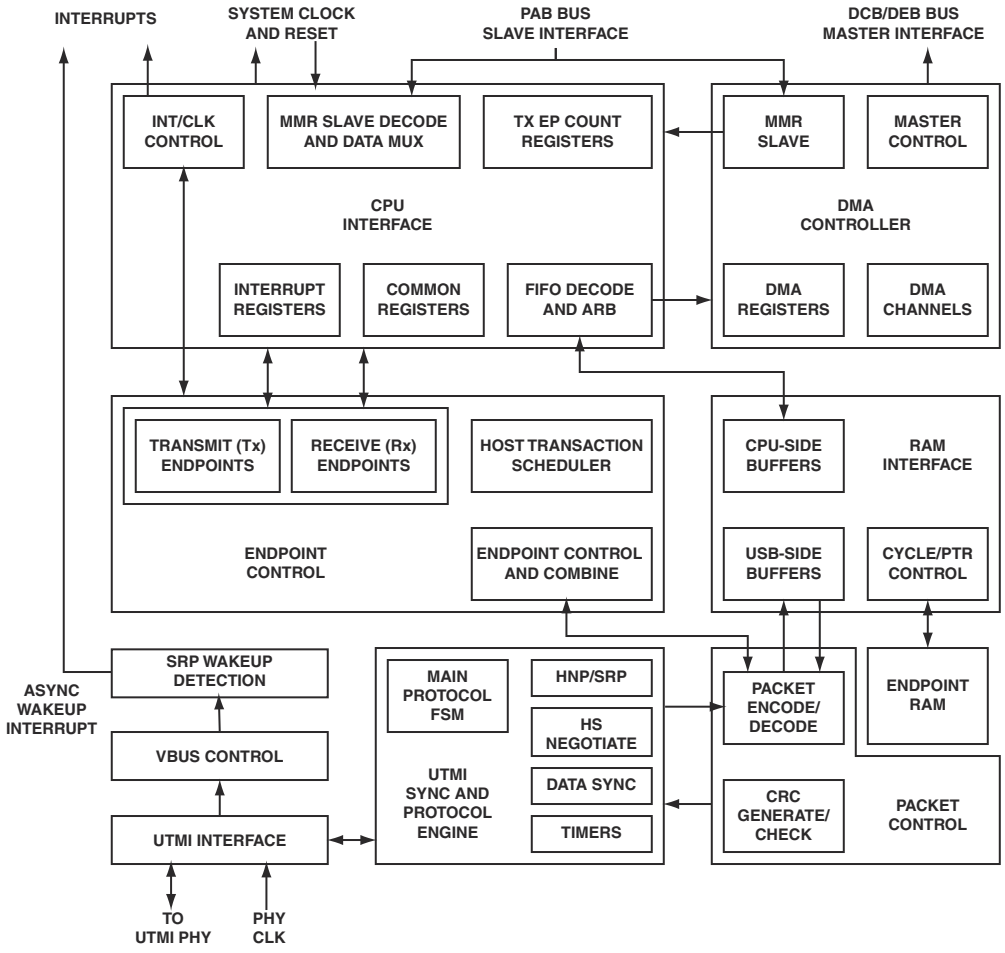


Figure 26-1. USB OTG Controller Block Diagram

The UTMI+ level 2 PHY interface provides a means of connecting a selection of high- or full-speed PHYs to the controller, from device-only PHYs through full OTG compliant PHYs. The details of the PHY interface can be found in "UTMI Interface" on page 26-56.

The USB controller requires a system clock frequency of greater than 30 MHz to operate correctly on USB.

The USB controller must not be used if the system clock is operating at a clock frequency below 30 MHz.

The USB controller is configured as either a USB OTG 'A' device or 'B' device depending on the type of plug inserted into its USB receptacle. This is determined by the state of the `USB_ID` (connector ID) pin.

The asynchronous wakeup circuit is used to detect when another 'B' device is asserting its D+ pull-up to initiate the SRP (session request protocol) when all other clocks are off. This circuit requires a slow clock (for example, 32kHz).

Before any endpoint register writes can be committed on endpoint zero, or before control transfers take place, the `GLOBAL_ENA` bit of the `USB_GLOBAL_CTL` register must be set in order to enable the system clock for the control logic. Likewise, before any endpoints can be set up and used to transfer data, the related control bit in the `USB_GLOBAL_CTL` register must be set.

Use of the controller for OTG functionality requires the capability to drive VBUS (as a default 'A' device powering the bus), to discharge VBUS (speeding up the time for VBUS to fall below the `SessionEnd` threshold as a 'B' device checking initial conditions), and to charge VBUS to 2.1V (when initiating SRP as a 'B' device). These controls are driven from the UTMI interface, but the controller also provides a separate interrupt register, `USB_OTG_VBUS_IRQ`, which represents the drive VBUS, discharge VBUS, and charge VBUS signaling. See "USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register" on page 26-136 for more information on these controls.

FIFO Configuration

Each bidirectional endpoint (provided as two unidirectional endpoints) has its own endpoint number (0 for control, 1–7 for data transfer). Although two endpoints might use the same number, the endpoints may support different transfer types. Each of these bidirectional endpoints has a fixed region of the SRAM in the USB controller to which it has access, and this feature dictates to some extent the types of transfers that may be used for that particular endpoint. This restriction follows from the maximum size of USB packets, which varies with each transfer type. Table 26-1 lists the endpoint FIFO configuration, with an indication of the transfer types possible for that particular buffer size.

Table 26-1. FIFO Sizes and Transfer Types

Bidirectional Endpoint (RX and TX)	FIFO Size (each direction)	USB Transfer Types
0	64 bytes	Size fixed for Control transfers.
1–4	128 bytes	Bulk, Interrupt, Isochronous
5–7	1024 bytes	Bulk, Interrupt, Isochronous

This configuration gives a total USB controller RAM size of 7232 bytes.

Each endpoint FIFO can buffer one or two packets (in double-buffered mode). The double buffered mode is automatically enabled when the software programs a maximum packet size for an endpoint that is equal to or less than half the actual FIFO size for that endpoint. Double-buffering is recommended for most applications to improve efficiency by reducing the frequency with which each endpoint needs to be serviced. Double-buffering Bulk transactions means that data transfer over the USB is not slowed if packets can be loaded/unloaded from the FIFO in the time it takes to transfer a packet over the bus. Double-buffering Isochronous transactions

also allows more time to load/unload the FIFO, but in addition, it also allows the SOF interrupt to be used to service the endpoint rather than the endpoint interrupt. This has the following advantages:

- easy detection of lost packets
- regular interrupt timing (making it easier to source/sink the data)
- If more than one Isochronous endpoint is used, they can all be serviced with one interrupt.

Interrupts

Three active-high top-level interrupts are provided from the USB controller: USB_INT0, USB_INT1 and USB_INT2. Each of these interrupts can be routed through the programming of a global mask register (USB_GLOBINTR) and can be sourced from control transfers, transmit (USB_INTRTX), and receive (USB_INTRRX) endpoint activity, from a range of conditions on the USB lines (USB_INTRUSB), or from requests for the USB controller to send VBUS control signals to an external analog chip (USB_OTG_VBUS_IRQ). The USB_INTRUSB and USB_OTG_VBUS_IRQ sources share the same interrupt line and can not be routed separately (for example, USB_INTRTX and USB_INTRRX). Finally, the DMA master channels use a separate interrupt, USB_DMAxINT, to indicate when a master transfer is pending.

Figure 26-2 shows the various sources of interrupts in the USB controller and how they are routed to the top-level interrupts using the USB_GLOBINTR register.

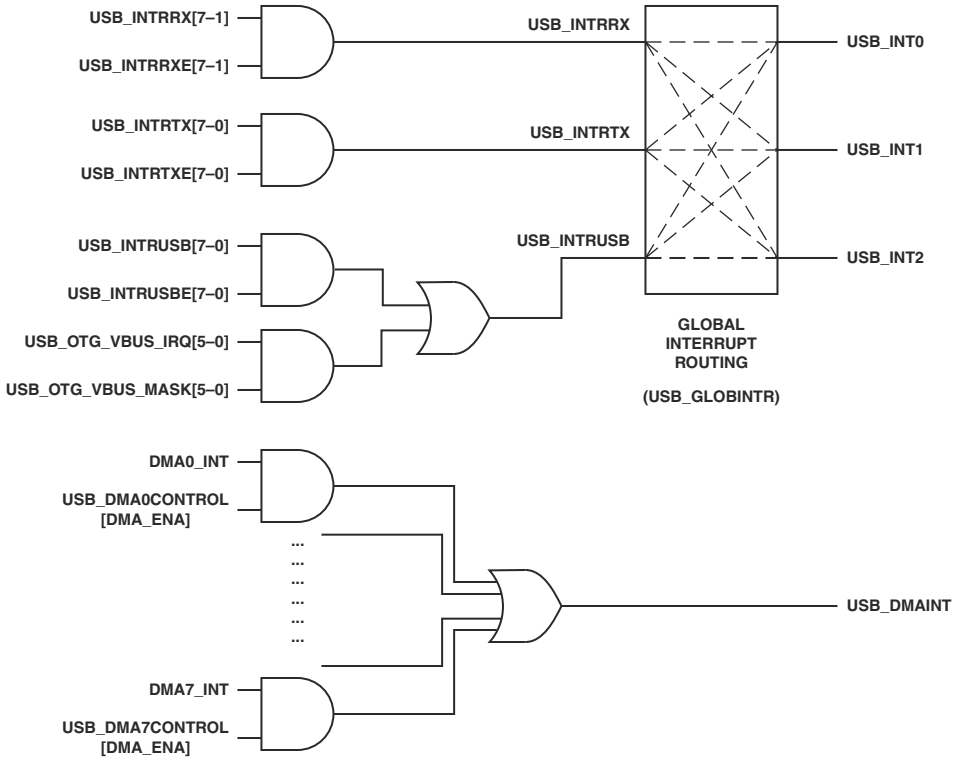


Figure 26-2. USB Interrupt Sources and Routing

Interrupts can be generated from control endpoint zero under the following conditions:

- When a control transaction ends before the end of the data is transferred.
- When a data packet is sent or received from the endpoint 0 FIFOs.

Interrupts can be generated from transmit endpoints (USB_INTRTX) under the following conditions:

- packet sent from the TX FIFO (host and peripheral mode)
- after three attempts at transmitting a packet with no valid handshake packet received (host mode)

Interrupts can be generated from receive endpoints (USB_INTRRX) under the following conditions:

- packet received into the RX FIFO (host and peripheral mode)
- when a STALL handshake is received (host mode)
- After three attempts at receiving a packet and no data packet is received (host mode).

Interrupts can be generated from the USB status (USB_INTRUSB) under the following conditions:

- When VBUS drops below the VBUS valid threshold during a session ('A' device only).
- When SRP signalling is detected ('A' device only).
- When device disconnect is detected (host mode).
- When a session ends (peripheral mode).
- Device connection detected (host mode).
- Start-of-frame (SOF)
- Reset signalling detected on USB (peripheral mode).
- Babble detected (host mode).

- In suspend mode when resume signalling detected on USB.
- When suspend signalling is detected (peripheral mode).

Interrupts are generated for the following VBUS control requests by the USB controller:

- drive VBUS greater than 4.4V (Default 'A' device)
- stop driving VBUS
- start charging VBUS (peripheral mode)
- stop charging VBUS
- start discharging VBUS (peripheral mode)
- stop discharging VBUS

Resets

The USB controller includes an active-high synchronous hardware reset sourced from the processor core. Another source of peripheral reset is through the USB, when USB reset signaling is detected on the I/O lines. As dictated by the USB 2.0 Specification, this state is entered when both the D+ and D– inputs are driven low for a period of 2.5 µs or more (though the reset itself is held for typically greater than 10ms by the USB host).

When a USB reset is detected, the USB controller performs the following actions:

- `USB_FADDR` register set to zero
- `USB_INDEX` register set to zero
- all endpoint FIFOs flushed
- all control and status registers cleared

- all interrupts enabled
- reset interrupt generated

The USB_INTRUSB, USB_OTG_VBUS_IRQ, USB_GLOBINTR, and USB_GLOBAL_CTL registers are *not* affected by the USB controller reset. These registers are only reset (along with those listed above) during a system reset.

Description of Operation

The USB OTG interface may operate in peripheral mode or host mode.

When the USB controller is operating in peripheral mode, the controller may be attached to a conventional host (such as a personal computer) or another OTG device operating in host mode. The second device can be high-speed or full-speed. When linked to another peripheral device, the USB controller can also act as the host, and if the other device is also a dual role controller, the two devices can switch roles as required.

The role taken by the USB controller depends on the way the devices are cabled together. Each USB cable has an 'A' and a 'B' device end. If the 'A' end of the cable is plugged into the device containing the USB controller, the USB controller takes the role of the host device and goes into host mode (in this case the HOST_MODE bit is set to 1). If the 'B' of the cable is plugged in, the USB controller goes instead into peripheral mode (and the HOST_MODE bit remains at 0).

When both devices contain dual role controllers, signaling may be used to switch the roles of the two devices, without switching the cable connecting the two devices. The conditions under which the USB controller may switch between peripheral and host mode are detailed in “Host Negotiation/Configuration” on page 26-82.

Peripheral Mode Operation

USB OTG interface operations for the peripheral mode differ from host mode in a number of ways. The following sections describe peripheral mode operations.

Endpoint Setup

In peripheral mode, there are a few endpoint-specific configuration bits that are used when setting up an endpoint for transfer for all types of peripheral transfer. They determine how the processor core interacts with the endpoint FIFO.

One key parameter required before transfer can occur through an endpoint is the maximum USB packet size that the endpoint can support. This value is set by the software and depends on a variety of system constraints. These include the size of hardware FIFO available and system latencies as well as the USB transfer type and class being used. The `USB_TX_MAX_PACKET` or `USB_RX_MAX_PACKET` defines the maximum amount of data that can be transferred to the selected endpoint in a single frame, and the value must match the programmed maximum individual packet size (*MaxPktSize*) of the standard endpoint descriptor for the endpoint. For TX endpoints, the maximum packet size is programmed using the `USB_TX_MAX_PACKET`. For RX endpoints, the `USB_RX_MAX_PACKET` register is used. The maximum packet size must not exceed the actual hardware endpoint FIFO size (see Table 26-1 on page 26-7). Because the USB controller uses a 16-bit interface, the value chosen for *MaxPktSize* should be an even number, as this selection simplifies transferring data between FIFOs and processor core.

If the size of the endpoint FIFO being used is at least twice the `USB_RX_MAX_PACKET` or `USB_TX_MAX_PACKET`, double buffering is automatically enabled for that endpoint.

Additional setup parameters are configured using the USB_RXCSR or USB_TXCSR register (depending on whether the endpoint in question is RX or TX). The DMA_ENA bit in this register is used to enable the assertion of the appropriate DMA request whenever the endpoint is able to receive or transmit another packet. The AUTOCLEAR_R and AUTOSET_R/T bits can be used to automatically set the FIFO ready triggers (RXPKTRDY and TXPKTRDY) whenever a packet is transferred to streamline DMA operation for transfers that span multiple packets. Refer to the descriptions in "USB OTG Registers" on page 26-97 for more details on the endpoint control and status registers.

IN Transactions as a Peripheral

When the USB controller is operating in peripheral mode, data for IN transactions is handled through the TX FIFOs. The maximum size of data packet that may be placed in a TX endpoint's FIFO for transmission is programmable and (where applicable) is determined by the value written to the USB_TX_MAX_PACKET register for that endpoint (maximum payload multiplied by the number of transactions per micro-frame).

The maximum packet size set for any endpoint must not exceed the FIFO size (see Table 26-1 on page 26-7).

The USB_TX_MAX_PACKET register should not be written-to while there is data in the FIFO, as unexpected results may occur.

If the size of the TX endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the USB_TX_MAX_PACKET register), only one packet can be buffered in the FIFO and *single packet buffering* is enabled. As each packet to be sent is loaded into the TX FIFO, the TXPKTRDY bit in USB_TXCSR needs to be set. If the AUTOSET_T bit in USB_TXCSR is set, the TXPKTRDY bit is automatically set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum, TXPKTRDY always has to be set manually (for example, set by the processor core).

When the TXPKTRDY bit is set, either manually or automatically, the FIFO_NOT_EMPTY_T bit in USB_TXCSR is also set and the packet is ready to be sent. When the packet is successfully sent, both TXPKTRDY and FIFO_NOT_EMPTY_T are cleared and the appropriate TX endpoint interrupt is generated (if enabled). The next packet can then be loaded into the FIFO.

If the size of the TX endpoint FIFO is at least twice the maximum packet size for this endpoint (as set in the USB_TX_MAX_PACKET), two packets can be buffered in the FIFO and *double packet buffering* is enabled. As each packet to be sent is loaded into the TX FIFO, the TXPKTRDY bit in USB_TXCSR needs to be set. If the AUTOSET_T bit in USB_TXCSR is set, the TXPKTRDY bit automatically is set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum, TXPKTRDY always has to be set manually (for example, set by the processor core). When the TXPKTRDY bit is set, either manually or automatically, the FIFO_NOT_EMPTY_T bit in USB_TXCSR also is set. TXPKTRDY is then immediately cleared (and an interrupt generated, if enabled). A second packet can now be loaded into the TX FIFO and TXPKTRDY set again (either manually or automatically if the packet is the maximum size). Both packets are now ready to be sent.

When the first packet is successfully sent, TXPKTRDY is cleared and the appropriate TX endpoint interrupt is generated (if enabled) to signal that another packet can now be loaded into the TX FIFO. The state of the FIFO_NOT_EMPTY_T bit at this point indicates how many packets may be loaded. If the FIFO_NOT_EMPTY_T bit is set then there is another packet in the FIFO and only one more packet can be loaded. If the FIFO_NOT_EMPTY_T bit is cleared then there are no packets in the FIFO and two more packets can be loaded.

OUT Transactions as a Peripheral

When the USB controller is operating in peripheral mode, data for OUT transactions is handled through the USB controller's RX FIFOs.

The maximum amount of data received by an RX endpoint in any frame or micro-frame (in high-speed mode) is programmable and is determined by the value written to the USB_EP_NIx_RXMAXP register for that endpoint. This is the maximum payload multiplied by the number of transactions per micro-frame (where applicable). The maximum packet size must not exceed the FIFO size (see Table 26-1 on page 26-7).

If the size of the RX endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the USB_RX_MAX_PACKET register), only one data packet can be buffered in the FIFO and single packet buffering is enabled. When a packet is received and placed in the RX FIFO, the RXPKTRDY bit and the FIFO_FULL_R bit in USB_RXCSR are set and the appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. After the packet is unloaded, the RXPKTRDY bit needs to be cleared in order to allow further packets to be received. If the AUTOCLEAR_R bit in USB_RXCSR is set and a maximum-sized packet is unloaded from the FIFO, the RXPKTRDY bit is cleared automatically. The FIFO_FULL_R bit is also cleared. For packet sizes less than the maximum, RXPKTRDY always has to be cleared manually (for example, set by the processor core).

If the size of the RX endpoint FIFO is at least twice the maximum packet size for the endpoint, two data packets can be buffered and *double packet buffering* is enabled. When the first packet to be received is loaded into the RX FIFO, the RXPKTRDY bit in USB_RXCSR is set and the appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. Note that the FIFO_FULL_R bit in USB_RXCSR is not set at this point. This bit is only set if a second packet is received and loaded into the RX FIFO.

After the first packet is unloaded, RXPKTRDY needs to be cleared in order to allow further packets to be received. If the AUTOCLEAR_R bit in USB_RXCSR is set and a maximum-sized packet is unloaded from the FIFO, the RXPKTRDY

bit is cleared automatically. For packet sizes less than the maximum, RXPKTRDY always has to be cleared manually (for example, set by the processor core).

If the FIFO_FULL_R bit was set to 1 when RXPKTRDY is cleared, the USB controller first clears the FIFO_FULL_R bit. The controller then sets RXPKTRDY again to indicate that there is another packet waiting in the FIFO to be unloaded.

Peripheral Transfer Workflows

The USB transfer types (control, bulk, isochronous and interrupt transfers) each have significantly different system requirements as well as individual USB transfer-specific features. This dictates that they are each dealt with slightly differently in software. For these reasons, there is no uniform way of doing transfers across all transfer types on the USB controller.

The following section provides some guideline peripheral mode transfer flows for each of the transfer types, in both IN (TX) and OUT (RX) directions. In the case of bulk endpoints, the optimal transfer flow differs depending on whether the final size of the transfer is known or unknown. Whether the transfer size is known or not depends on the USB driver class being used. Some define the complete transfer size, and others operate on a packet-by-packet basis using a short packet (a packet of less than USB_TX_MAX_PACKET or less than USB_RX_MAX_PACKET) to denote the end of a transfer.

Each of the workflows use the following common method.

1. Configure the endpoint control and status registers and the USB_TX_MAX_PACKET or USB_RX_MAX_PACKET value.
2. Configure the appropriate data transfer mechanism (DMA or interrupt setup).
3. Data transfer phase

The workflows do not describe the USB controller's actions immediately preceding the endpoint setup (for example, the reception of an IN/OUT token from the host, token validity checking, or NAK generation, among others). Note also that there is currently no error-handling contained in the workflows (for example, checking `FIFO_FULL_R` bit before writing data).

The terms packets, frames and transfers are used in the proceeding sections with their strict USB definitions. (See the "Glossary of USB Terms " on page 26-148 for these definitions.)

Control Transactions as a Peripheral

Endpoint 0 is the main control endpoint of the USB controller. As such, the routines required to service Endpoint 0 are more complicated than those required to service other endpoints.

The software is required to handle all the Standard Device Requests that may be sent or received through Endpoint 0. These are described in *Universal Serial Bus Specification*, Revision 2.0, Chapter 9. The protocol for these device requests involves different numbers and types of transactions per transfer. To accommodate this, the processor needs to take a state machine approach to command decoding and handling.

The Standard Device Requests received by a USB peripheral can be divided into three categories: Zero Data Requests (in which all the information is included in the command), Write Requests (in which the command will be followed by additional data), and Read Requests (in which the device is required to send data back to the host).

This section looks at the sequence of actions that the software must perform to process these different types of device request.

The Setup packet associated with a Standard Device Request should include an 8-byte command. A Setup packet containing a command field of anything other than 8 bytes will be automatically rejected by the USB controller.

Write Requests

Write requests involve an additional packet (or packets) of data being sent from the host after the 8-byte command. An example of a 'Write' Standard Device Request is: SET_DESCRIPTOR.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The RxPktRdy bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded.

As with a zero data request, the USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command is read from the FIFO) but in this case the DataEnd bit should not be set (indicating that more data is expected).

When a second Endpoint 0 interrupt is received, the USB_CSR0 register is read to check the endpoint status. The RxPktRdy bit is set to indicate that a data packet is received. The USB_COUNT0 register should then be read to determine the size of this data packet. The data packet can then be read from the Endpoint 0 FIFO.

If the length of the data associated with the request (indicated by the wLength field in the command) is greater than the maximum packet size for Endpoint 0, further data packets will be sent. In this case, USB_CSR0 is written to set the ServicedRxPktRdy bit, but the DataEnd bit should not be set.

When all the expected data packets have been received, the USB_CSR0 register is written to set the ServicedRxPktRdy bit and to set the DataEnd bit (indicating that no more data is expected).

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the USB_CSR0 register should be written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host sends more data, the USB controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host sends more data after the DataEnd has been set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

Read Requests

Read requests have a packet (or packets) of data sent from the function to the host after the 8-byte command. Examples of Standard Device Requests for Read are: GET_CONFIGURATION, GET_INTERFACE, GET_DESCRIPTOR, GET_STATUS, SYNCH_FRAME.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The RxPktRdy bit in USB_CSR0 will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded. The USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command has read from the FIFO).

The data to be sent to the host should then be written to the Endpoint 0 FIFO. If the data to be sent is greater than the maximum packet size for Endpoint 0, only the maximum packet size should be written to the FIFO. The USB_CSR0 register should then be written to set the TxPktRdy bit (indicating that there is a packet in the FIFO to be sent). When the packet has been sent to the host, another Endpoint 0 interrupt will be generated and the next data packet can be written to the FIFO.

When the last data packet has been written to the FIFO, the USB_CSR0 register should be written to set the TxPktRdy bit and to set the DataEnd bit (indicating that there is no more data after this packet).

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the USB_CSR0 register should be written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host requests data, the USB controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host requests more data after DataEnd has been set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

Zero Data Requests

Zero data requests have all their information included in the 8-byte command and require no additional data to be transferred.

Examples of zero data Standard Device Requests are: SET_FEATURE, CLEAR_FEATURE, SET_ADDRESS, SET_CONFIGURATION, SET_INTERFACE.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The RxPktRdy bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO, decoded and the appropriate action taken. For example if the command is SET_ADDRESS, the 7-bit address value contained in the command is written to the USB_FADDR register.

When the host moves to the status stage it still addresses the device with the default address, therefore the USB_FADDR should not be written before the host moves to the status stage. In the next transaction the host will then use this new address to address the device.

The USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command is read from the FIFO) and to set the DataEnd bit (indicating that no further data is expected for this request).

When the host moves to the status stage of the request, a second Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the second interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it is decoded, the USB_CSR0 register is written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host moves to the status stage of the request, the USB controller will send a STALL to tell the host that the request was not executed. A second Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host sends more data after the DataEnd bit is set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

ENDPOINT 0 States

When the USB is operating as a peripheral, the Endpoint 0 control needs three modes (IDLE, TX and RX shown in Figure 26-3) corresponding to the different phases of the control transfer and the states Endpoint 0 enters for the different phases of the transfer (see "Endpoint 0 Service Routine as Peripheral" on page 26-24).

The default mode on power-up or reset should be IDLE. The RxPktRdy bit becoming set when Endpoint 0 is in IDLE state indicates a new device request. Once the device request is unloaded from the FIFO, the USB decodes the descriptor to find whether there is a data phase and, if so, the direction of the data phase of the control transfer (in order to set the FIFO direction).

Depending on the direction of the data phase, Endpoint 0 goes into either TX state or RX state. If there is no data phase, Endpoint 0 remains in IDLE state to accept the next device request.

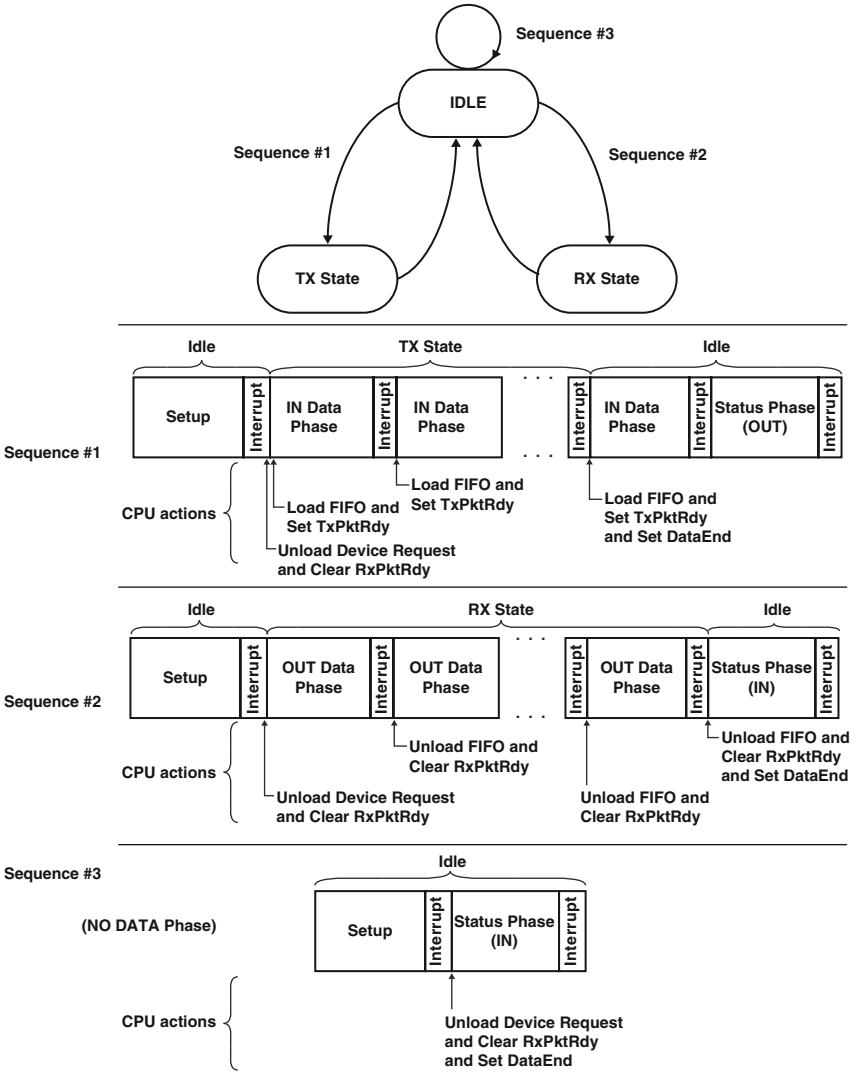


Figure 26-3. Endpoint 0 Control States

The processor needs to take different actions at the different phases of the possible transfers (for example, “Loading the FIFO”, “Setting TxPktRdy”) are indicated in Figure 26-3. Note that the USB changes the FIFO direction depending on the direction of the data phase, independently of the processor.

Endpoint 0 Service Routine as Peripheral

An Endpoint 0 interrupt is generated:

- When the USB controller sets the RxPktRdy bit after a valid token has been received and data has been written to the FIFO.
- When the USB controller clears the TxPktRdy bit after the data packet in the FIFO has been successfully transmitted to the host.
- When the USB controller sets the SentStall bit after a control transaction is ended due to a protocol violation.
- When the USB controller sets the SetupEnd bit because a control transfer has ended before DataEnd is set.

The bits mentioned above, are in the USB_CSR0 register.

Whenever the Endpoint 0 service routine is entered, the firmware must first check whether the current control transfer has been ended due to either a STALL condition or a premature end-of-control transfer. If the control transfer ends due to a STALL condition, the SentStall bit would be set. If the control transfer ends due to a premature end-of-control transfer, the SetupEnd bit would be set. In either case, the firmware should abort processing the current control transfer and set the state to IDLE.

Once the firmware has determined that the interrupt was not generated by an illegal bus state, the next action depends on the Endpoint state.

If Endpoint 0 is in IDLE state, the only valid reason an interrupt can be generated is as a result of the core receiving data from the USB bus. The service routine must check for this by testing the RxPktRdy bit. If this bit is

set, then the core has received a SETUP packet. This must be unloaded from the FIFO and decoded to determine the action the core must take. Depending on the command contained within the SETUP packet, Endpoint 0 will enter one of three states:

- If the command is a single packet transaction (`SET_ADDRESS`, `SET_INTERFACE` etc.) without a data phase, the endpoint will remain in IDLE state.
- If the command has an OUT data phase (`SET_DESCRIPTOR` etc.), the endpoint will enter RX state.
- If the command has an IN data phase (`GET_DESCRIPTOR` etc.), the endpoint will enter TX state.

If the endpoint is in TX state, the interrupt indicates that the core has received an IN token and data from the FIFO has been sent. The firmware must respond to this either by placing more data in the FIFO if the host is still expecting more data[1] or by setting the `DataEnd` bit to indicate that the data phase is complete. Once the data phase of the transaction has been completed, Endpoint 0 should be returned to IDLE state to await the next control transaction.

If the endpoint is in RX state, the interrupt indicates that a data packet has been received. The firmware must respond by unloading the received data from the FIFO. The firmware must then determine whether it has received all of the expected data[1]. If it has, the firmware should set the `DataEnd` bit and return Endpoint 0 to IDLE state. If more data is expected, the firmware should set the `ServicedRxPktRdy` bit to indicate that it has read the data in the FIFO and leave the endpoint in RX state.

[1] Command transactions all include a field that indicates the amount of data the host expects to receive or is going to send.

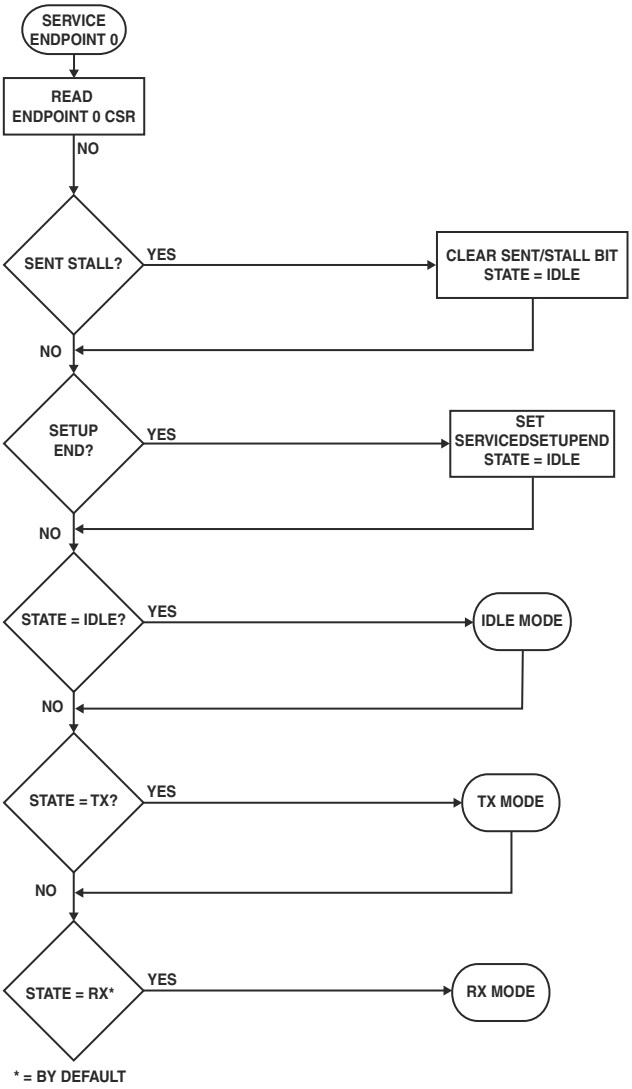


Figure 26-4. Endpoint 0 Service Routine

Idle Mode

The Endpoint 0 control must select the IDLE mode at power-on or reset. The Endpoint 0 control should return to this mode when the RX and TX modes are terminated.

This is also the mode in which the SETUP phase of control transfer is handled (see Figure 26-5 on page 26-27).

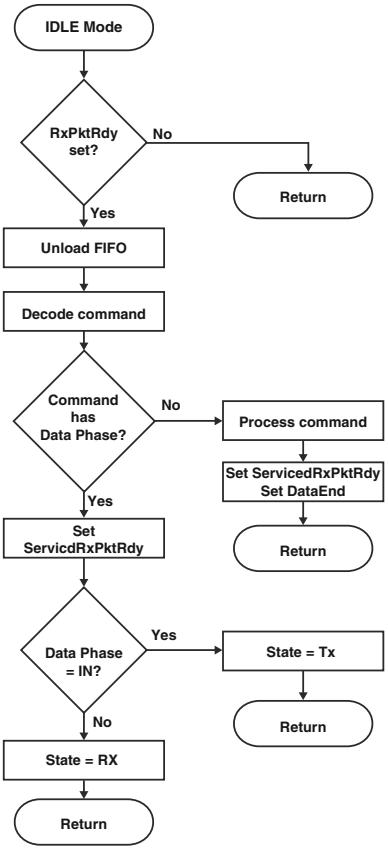


Figure 26-5. Endpoint 0 Idle Mode (Setup Phase)

TX Mode

When the endpoint is in TX state, all arriving IN tokens need to be treated as part of a data phase until the required amount of data has been sent to the host. If either a SETUP or an OUT token is received while the endpoint is in the TX state, a `SetupEnd` condition would occur since the core expects only IN tokens.

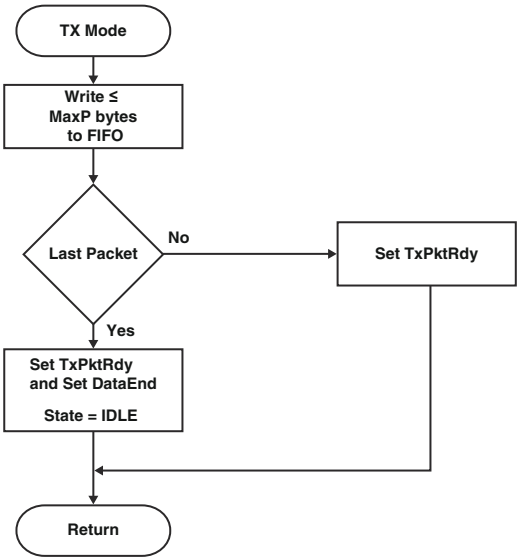


Figure 26-6. Endpoint 0 TX Mode

Three events can cause the TX mode to terminate before the expected amount of data has been sent:

- The host sends an invalid token causing a `SetupEnd` bit set.
- The firmware sends a packet containing less than the maximum packet size for Endpoint 0.
- The firmware sends an empty data packet.

Until the transaction is terminated, when the firmware receives an interrupt which indicates that a packet has been sent from the FIFO, it simply loads the FIFO. An interrupt is generated when TxPktRdy is cleared.

When the firmware forces the termination of a transfer (by sending a short or empty data packet), it should set the `DataEnd` bit to indicate to the core that the data phase is complete and that the core should receive an acknowledge packet next.

RX Mode

In RX mode, all arriving data should be treated as part of a data phase until the expected amount of data has been received. If either a SETUP or an IN token is received while the endpoint is in RX state, a `SetupEnd` condition would occur since the core expects only OUT tokens.

Three events can cause the RX mode to terminate before the expected amount of data has been received:

- The host sends an invalid token causing a `SetupEnd` bit set.
- The host sends a packet which contains less than the maximum packet size for Endpoint 0.
- The host sends an empty data packet.

Until the transaction is terminated, when the firmware receives an interrupt which indicates that new data has arrived (`RxPktRdy` bit set), it simply needs to unload the FIFO and clear `RxPktRdy` by setting the `ServicedRxPktRdy` bit.

When the firmware detects the termination of a transfer (by receiving either the expected amount of data or an empty data packet), it should set the `DataEnd` bit to indicate to the core that the data phase is complete and that the core should receive an acknowledge packet next.

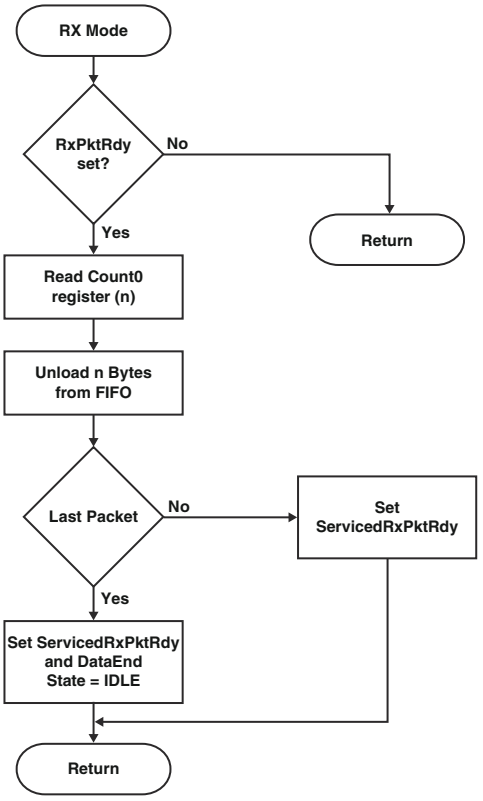


Figure 26-7. Endpoint 0 RX Mode

If the last packet is not a multiple of four bytes it is strongly recommended that the remainder (n bytes mod 4) be unloaded from the FIFO using the byte addressing FIFO register (EP0 FIFO address + 4). This will prevent the USB controller from sending non-null data during the status phase of the control transfer.

Peripheral Mode, Bulk IN, Transfer Size Known

For this process, the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes, must be known.

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOSET_T = 1, ISO_T = 0, FRCDATATOG = 0 in USB_TXCSR.
3. Load *TxferSize* into USB_TXCOUNT.
4. Configure the DMA controller to write full *TxferSize*/2 half words into the corresponding TX FIFO address.
5. On each USB_DMAxINT transition, the DMA controller writes a new packet into the FIFO. TXPKTRDY is automatically set when each new packet is written.
6. Step 5 is repeated for each full packet of the transfer.
7. Even if the final packet is a short packet, the packet automatically is detected by the USB controller (because USB_TXCOUNT is zero) and TXPKTRDY is set.

Peripheral Mode, Bulk IN, Transfer Size Unknown

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is assumed to be an even number of bytes.

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOSET_T = 1, ISO_T = 0, FRCDATATOG = 0 in USB_TXCSR.
3. Configure the DMA controller to write *MaxPktSize*/2 half words into the corresponding TX FIFO address on each USB_DMAxINT.

4. Set up an ISR, sensitive to the DMA work-block-complete interrupt, that writes a remaining short packet into the TX FIFO using processor core DMA. Then set TXPKTRDY or simply send a zero-length packet by toggling TXPKTRDY.
5. On each USB_DMAxINT transition, the DMA controller writes a new packet into the FIFO. TXPKTRDY automatically is set when each new packet is written.
6. Step 5 is repeated for each full packet of the transfer.
7. The final short/zero-length packet is managed by the ISR from step 4.

Peripheral Mode, ISO IN, Small *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is less than 128 bytes and is an even number of bytes. Double buffering is assumed to be enabled, and the auto set feature unused (because packets are often less than *MaxPktSize*).

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set ISO_T = 1 in USB_TXCSR.
3. Preload the first two packets into the endpoint TX FIFO and set TXPKTRDY (or alternatively use the USB_TXCOUNT feature that sets TXPKTRDY after USB_TXCOUNT bytes have been loaded).
4. Set up an ISR, sensitive to the SOF_B interrupt, which writes a new packet into the TX FIFO and sets TXPKTRDY.
5. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
6. Step 5 is repeated for each ISO packet.

Peripheral Mode, ISO IN, Large *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is greater than 128 bytes and is an even number of bytes. Double buffering is assumed to be enabled, and the auto set feature unused (because packets are often less than *MaxPktSize*).

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set ISO_T = 1 in USB_TXCSR.
3. Set ISO_UPDATE = 1 in USB_POWER to prevent initial packet loaded into the FIFO from being transmitted on USB until the next 1ms frame.
4. Load the total number of bytes for the first two packets into USB_TXCOUNT.
5. Configure the DMA controller to pre-load the two packets (as half words) into the corresponding TX FIFO address. TXPKTRDY automatically is set by the USB controller when USB_TXCOUNT bytes have been loaded.
6. Set up an ISR, sensitive to the SOF_B interrupt, which writes a new packet into the TX FIFO by loading USB_TXCOUNT with the size of the packet, then configuring the DMA controller to load the packet.
7. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
8. Step 7 is repeated for each ISO packet.

Peripheral Mode, Bulk OUT, Transfer Size Known

For this process, the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes must be known.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOCLEAR_R = 1, ISO_R = 0, FRCDATATOG = 0, DMAREQMODE_R = 0 in USB_RXCSR.
3. Configure the DMA controller to read the full *TxferSize*/2 half words from the corresponding RX FIFO address.
4. On each USB_DMAxINT transition, the DMA controller reads another packet from the FIFO. RXPKTRDY is automatically cleared by the USB controller when each new packet is read.
5. Step 5 is repeated for each full packet of the transfer.
6. If *TxferSize* is not an exact multiple of *MaxPktSize*, the final USB_DMAxINT transition causes the DMA controller to read out only the short packet that remains.

Peripheral Mode, Bulk OUT, Transfer Size Unknown

For this process, the maximum individual packet size (*MaxPktSize*) in bytes must be known.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOCLEAR_R = 1, ISO_R = 0, FRCDATATOG = 0, DMAREQMODE_R = 1 in USB_RXCSR.
3. Set the appropriate EPx_RX_E bit in USB_INTRRXE.
4. Configure the DMA controller to read *MaxPktSize*/2 half words from the corresponding RX FIFO address on each USB_DMAxINT transition.

5. Set up an ISR, sensitive to the RX interrupt, which reads USB_RXCOUNT and then transfers USB_RXCOUNT bytes (in half words) from the RX FIFO to the processor core. Depending on the number of bytes in the FIFO, this can be performed by configuring the DMA to read the data, or by reading it with the processor core.
6. On each USB_DMAxINT transition, the DMA controller reads a packet from the FIFO. RXPKTRDY is automatically cleared by the USB controller when each new packet is read.
7. Step 5 is repeated for each full packet of the transfer.
8. If a packet is received that is less than *MaxPktSize*, the RX interrupt goes high, and the ISR from step 5 reads out the remaining short packet.

Peripheral Mode, ISO OUT, Small *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is less than 128 bytes, and double buffering is assumed to be enabled.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set ISO_R = 1 in USB_RXCSR.
3. Set up an ISR, sensitive to the SOF_B interrupt, that reads the FIFO_FULL_R bit, reads the USB_RXCOUNT status register, and finally removes one or two packets (equal to the USB_RXCOUNT number of bytes) from the FIFO then clears RXPKTRDY.
4. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
5. Step 4 is repeated for each ISO packet.

Peripheral Mode, ISO OUT, Large *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is greater than 128 bytes, and double buffering is assumed to be enabled.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set ISO_R = 1 in USB_RXCSR.
3. Set up an ISR, sensitive to the SOF_B interrupt, that reads the FIFO_FULL_R bit, reads the USB_RXCOUNT status register, and finally configures the DMA controller to remove one or two packets (equal to the USB_RXCOUNT number of bytes) from the FIFO.
4. Set up an ISR, sensitive to the DMA work-block-complete interrupt to clear RXPKTRDY.
5. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
6. Step 5 is repeated for each ISO packet.

Peripheral Mode Suspend

When no activity has occurred on the USB for 3 ms, the USB controller enters suspend mode. If the suspend interrupt (SUSPEND_B) is enabled, an interrupt is generated at this time.

When resume signaling is detected, the USB controller exits suspend mode. If the RESUME_B interrupt is enabled, an interrupt is generated. The processor core can also force the USB controller to exit suspend mode by setting the RESUME_MODE bit in the USB_POWER register. When this bit is set, the USB controller exits suspend mode and drives resume signaling onto the bus. The processor core should clear this bit after 10 ms (a maximum of 15 ms) to end resume signaling.

No RESUME_B interrupt is generated when suspend mode is exited by the processor core.

Start-of-frame (SOF) Packets

When the USB controller is operating in peripheral mode, it should receive a start-of-frame packet from the host every millisecond when in full-speed mode, or every 125 microseconds when in high-speed mode.

When the SOF packet is received, the 11-bit frame number contained in the packet is written into the USB_FRAME register and an output pulse, lasting one USB clock bit period, is generated on SOF_PULSE (internal USB controller signal). A SOF_B interrupt is also generated (if enabled in the USB_INTRUSBE register).

After the USB controller has started to receive SOF packets, the controller expects one every millisecond (or 125 µs when in high-speed mode). If no SOF packet is received after 1.00358 ms (or 125.125 µs), it is assumed that the packet is lost. An SOF_PULSE (together with a SOF_B interrupt, if enabled) is still generated though the USB_FRAME register is not updated. The USB controller continues to generate an SOF_PULSE every millisecond (or 125 µs) and re-synchronizes these pulses to the received SOF packets when these packets are successfully received again.

Soft Connect/Soft Disconnect

In peripheral mode, the USB controller can be programmed to switch between normal mode and non-driving mode by setting or clearing the SOFT_CONN bit of the USB_POWER register. When this SOFT_CONN bit is set to 1, the USB controller is placed in its normal mode and the D+/D– lines of the USB bus are enabled. When the SOFT_CONN bit is zero, the PHY is put into non-driving mode and D+ and D– are three-stated. The USB controller appears to have been disconnected from the USB bus.

After system reset, SOFT_CONN is cleared to 0. From that point, the USB controller appears disconnected until the software has set SOFT_CONN to 1. The application software can then choose when to set the PHY to its normal mode. Systems with a lengthy initialization procedure may use this to ensure that initialization is complete and the system is ready to perform enumeration before connecting to the USB.

Error Handling As a Peripheral

A control transfer may be aborted due to a protocol error on the USB, the host prematurely ending the transfer, or if the function controller software wishes to abort the transfer (for example, because it cannot process the command).

The USB controller will automatically detect protocol errors and send a STALL packet to the host under the following conditions:

1. The host sends more data during the OUT data phase of a write request than was specified in the command. This condition is detected when the host sends an OUT token after the DataEnd bit is set.
2. The host requests more data during the IN data phase of a read request than was specified in the command. This condition is detected when the host sends an IN token after the DataEnd bit in the USB_CSR0 register is set.
3. The host sends more than *MaxPktSize* data bytes in an OUT data packet.
4. The host sends a non-zero length DATA1 packet during the status phase of a read request.

When the USB controller has sent the STALL packet, it sets the SentStall bit and generates an interrupt. When the software receives an Endpoint 0 interrupt with the SentStall bit set, it should abort the current transfer, clear the SentStall bit, and return to the IDLE state.

If the host prematurely ends a transfer by entering the status phase before all the data for the request is transferred, or by sending a new SETUP packet before completing the current transfer, then the SetupEnd bit will be set and an Endpoint 0 interrupt generated. When the software receives an Endpoint 0 interrupt with the SetupEnd bit set, it should abort the current transfer, set the ServicedSetupEnd bit, and return to the IDLE state. If the RxPktRdy bit is set, this indicates that the host has sent another SETUP packet and the software should then process this command.

If the software wants to abort the current transfer, because it cannot process the command or has some other internal error, then it should set the SendStall bit. The USB controller will then send a STALL packet to the host, set the SentStall bit and generate an Endpoint 0 interrupt.

Stalls Issued to Control Transfers

In peripheral mode, the USB controller automatically issues a STALL handshake to a control transfer under the following conditions:

1. The host sends more data during an OUT data phase of a control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB controller when the host sends an OUT token (instead of an IN token) after the processor core has unloaded the last OUT packet and set DATAEND.
2. The host requests more data during an IN data phase of a control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB controller when the host sends an IN token (instead of an OUT token) after the processor core has cleared TXPKTRDY and set DATAEND in response to the ACK issued by the host to what should have been the last packet.
3. The host sends more than *MaxPktSize* data with an OUT data token.

4. The host sends the wrong PID (packet identifier) for the OUT status phase of a control transfer.
5. The host sends more than a zero length data packet for the OUT status phase.

Zero Length OUT Data Packets in Control Transfers

A zero-length OUT data packet is used to indicate the end of a control transfer. In normal operation, such packets should only be received after the entire length of the device request is transferred (for example, after the processor core has set `DATAEND`). If the host sends a zero-length OUT data packet before the entire length of device request is transferred, this packet signals the premature end of the transfer. In this case, the USB controller automatically flushes any IN token loaded by processor core ready for the data phase from the FIFO and sets `SETUPEND`.

Host Mode Operation

USB OTG interface operations in host mode differ from peripheral mode in a number of ways. The following sections describe host mode operations.

Endpoint Setup and Data Transfer

When the `HOST_MODE` bit is set to 1, the USB controller operates as a host for point-to-point communications with another USB device. This second device may be either a high-speed, full-speed, or low-speed USB function, but it may not be a hub. Control, bulk, isochronous or interrupt transactions are supported between the USB controller and the second device.

Transfers between the subsystem and endpoint FIFOs in host mode are similar to peripheral mode. With this in mind, see many of the descriptions of processor core to FIFO data transfer in "Peripheral Mode Operation" on page 26-13.

Control Transaction as a Host

Host control transactions are conducted through Endpoint 0. The software is required to handle all the Standard Device Requests that may be sent or received through Endpoint 0 (as described in *Universal Serial Bus Specification*, Revision 2.0, Chapter 9).

For a USB peripheral, there are three categories of Standard Device Requests to be handled: Zero Data Requests (in which all the information is included in the command), Write Requests (in which the command will be followed by additional data), and Read Requests (in which the device is required to send data back to the host).

Zero Data Requests comprise a SETUP command followed by an IN status phase.

Write Requests comprise a SETUP command, followed by an OUT data phase followed by an IN status phase.

Read Requests comprise a SETUP command, followed by an IN data phase followed by an OUT status phase.

A timeout may be set to limit the length of time during which the USB controller will retry a transaction that is continually NAKed by the target. This limit can be between 2 and 2^{15} frames/microframes and is set through the `USB_NAKLIMITO` register.

The following sections look at the steps in different phases of a control transaction to describe the actions of the core in issuing Standard Device Requests.

Before initiating transactions as a host, the `USB_FADDR` register needs to be set to address the peripheral device. When the device is first connected, `USB_FADDR` is set to zero. After a `SET_ADDRESS` command is issued, `USB_FADDR` is set to the target's new address.

Setup Phase as a Host

The processor core driving the host device performs the following actions for the SETUP phase of a control transaction.

1. Load the eight bytes of the required device request command into the Endpoint 0 FIFO
2. Set SETUPPKT_H (bit 3)and TxPtRdy (bit 1) of the USB_CSR0 register. These bits must be set together.

   The USB controller then sends a SETUP token followed by the 8-byte command to Endpoint 0 of the addressed device, retrying as necessary.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt (for example, set IntrTx.D0). The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit or the NAK_TIMEOUT_H bit is set.

   If STALL_RECEIVED_H is set, it indicates that the target did not accept the command (for example, because it is not supported by the target device) and so has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the SETUP packet and the following data packet three times without getting a response.

   If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the SETUP packet, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

4. If none of STALL_RECEIVED_H, ERROR_H or NAK_TIMEOUT_H is set, the SETUP phase is correctly acknowledged and the processor core should proceed to the following IN data phase, OUT data phase or IN status phase specified for the particular Standard Device Request.

IN Data Phase as a Host

The processor core driving the host device performs the following actions for the IN data phase of a control transaction.

1. Set REQPKT_H in USB_CSR0.
2. Wait while the USB controller sends the IN token and then receives the required data back.
3. When the USB controller generates the Endpoint 0 interrupt (for example, by setting EP0_TX in the USB_INTRTX register)—read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit, the NAK_TIMEOUT_H bit or RxPktRdy is set.

   If STALL_RECEIVED_H is set, it indicates that the target has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the required IN token three times without getting a response.

   If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit; or to abort the transaction by clearing REQPKT_H before clearing the NAK_TIMEOUT_H bit.

4. If RxPktRdy is set, the processor core should read the data from the Endpoint 0 FIFO, then clear RxPktRdy.
5. If further data is expected, the processor core should repeat the previous steps.

   When all the data is successfully received, the processor core should proceed to the OUT status phase of the control transaction.

OUT Data as a Host (Control)

The processor core driving the host device performs the following actions for the OUT data phase of a control transaction.

1. Load the data to be sent into the Endpoint 0 FIFO
2. Set the TxPtRdy bit in USB_CSR0.

   The USB controller then proceeds to send an OUT token followed by the data from the FIFO to Endpoint 0 of the addressed device, retrying as necessary.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt (for example, set EP0_TX in USB_INTRTX register). The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit (D2), the ERROR_H bit (D4) or the NAK_TIMEOUT_H bit (D7) is set.

   If STALL_RECEIVED_H is set, it indicates that the target has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the OUT token and the following data packet three times without getting a response.

   If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the OUT token, for longer than the time set in the USB_NAKLIMIT0 register. The

USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit; or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

If none of STALL_RECEIVED_H, ERROR_H or USB_NAKLIMITO is set, the OUT data is correctly acknowledged.

4. If further data needs to be sent, the processor core should repeat the previous steps.

   When all the data is successfully sent, the processor core should proceed to the IN status phase of the control transaction.

IN Status Phase as a Host (Following SETUP Phase or OUT Data Phase)

The processor core driving the host device performs the following actions for the IN status phase of a control transaction.

1. Set STATUSPKT_H_H and REQPKT_H (bits 6 and 5 of USB_CSR0, respectively). These bits must be set together.
2. Wait while the USB controller both sends an IN token and receives a response from the USB peripheral.
3. When the USB controller generates the Endpoint 0 interrupt (for example, sets EP0_TX in USB_INTRTX register), read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit, the NAK_TIMEOUT_H bit or RxPktRdy is set.

   If STALL_RECEIVED_H is set, it indicates that the target could not complete the command and so has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the required IN token three times without getting a response.

If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by clearing REQPKT_H and STATUSPKT_H_H before clearing the NAK_TIMEOUT_H bit.

4. If RxPktRdy is set, the processor core should simply clear RxPktRdy.

OUT Status Phase as a Host (following IN Data Phase)

The processor core driving the host device performs the following actions for the OUT status phase of a control transaction.

1. Set STATUSPKT_H and TxPktRdy bits. These bits must be set together.
2. Wait while the USB controller both sends the OUT token and a zero-length DATA1 packet.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt. The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit or the NAK_TIMEOUT_H bit is set.

   If STALL_RECEIVED_H is set, it indicates that the target could not complete the command and so has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the STATUS packet and the following data packet three times without getting a response.

If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

4. If none of STALL_RECEIVED_H, ERROR_H or NAK_TIMEOUT_H is set, the status phase is correctly acknowledged.

Host IN Transactions

When the USB controller operates as a host, IN transactions are handled like OUT transactions are handled when the USB controller is operating as a peripheral. But the transaction must first be initiated by setting the REQPKT_H bit in USB_RXCSR. This bit indicates to the transaction scheduler that there is an active transaction on this endpoint. The transaction scheduler then sends an IN token to the target function.

When the packet is received and placed in the RX FIFO, the RXPKTRDY bit in USB_RXCSR is set, and the appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. When the packet is unloaded, RXPKTRDY is cleared. The AUTOCLEAR_R bit in the USB_RXCSR register can be used to have RXPKTRDY automatically cleared when a maximum sized packet is unloaded from the FIFO. There is also an AUTOREQ_RH bit in USB_RXCSR that causes the REQPKT_H bit to be automatically set when the RXPKTRDY bit is cleared. The AUTOCLEAR_R and AUTOREQ_RH bits can be used with an external DMA controller to perform complete bulk transfers without processor core intervention.

If the target function responds to a bulk or interrupt IN token with a NAK, the USB controller keeps retrying the transaction until the NAK limit set (in USB_NAKLIMIT0) is reached. If the target function responds with a STALL, the USB controller does not retry the transaction, but does interrupt the processor core with the RXSTALL_TH bit in the USB_RXCSR register set. If the target function does not respond to the IN token within the required time (or there was a CRC or bit-stuff error in the packet), the USB controller retries the transaction. If after three attempts the target function still has not responded, the USB controller clears the REQPKT_H bit and interrupts the processor core with the DATAERROR_R bit in USB_RXCSR set.

Host OUT Transactions

When the USB controller operates as a host, OUT transactions are handled in a similar manner to the way IN transactions are handled when the USB controller operates as a peripheral.

The TXPKTRDY bit in the USB_TXCSR register needs to be set as each packet is loaded into the TX FIFO and the AUTOSET_T bit in USB_TXCSR can be used to cause the TXPKTRDY bit to be automatically set when a maximum sized packet is loaded into the FIFO. The AUTOSET_T bit can be used with an external DMA controller to perform complete bulk transfers without processor core intervention.

If the target function responds to the OUT token with a NAK, the USB controller keeps retrying the transaction until the NAK limit set in USB_NAKLIMIT0 is reached. If the target function responds with a STALL, the USB controller does not retry the transaction, but does interrupt the processor core with the RXSTALL_TH bit in the USB_TXCSR register set. If the target function does not respond to the OUT token within the required time (or there was a CRC or bit-stuff error in the packet), the USB controller retries the transaction. If after three attempts the target function still has not responded, the USB controller flushes the FIFO and interrupts the processor core with the ERROR_TH bit in USB_TXCSR set.

Transaction Scheduling

When operating as a host, the USB controller maintains a frame counter. If the target function is a full-speed device, the USB controller automatically sends an SOF packet at the start of each frame or micro-frame. If the target function is a low-speed device, a K state is transmitted on the bus to act as a *keep-alive* to stop the low-speed device from going into suspend mode.

After the SOF packet is transmitted, the USB controller cycles through all the endpoints looking for active transactions. An active transaction is defined as an RX endpoint for which the REQPKT_H bit is set or a TX endpoint for which the TXPKTRDY bit is set. An active isochronous or interrupt transaction will only start if it is found on the first transaction scheduler cycle of a frame and if the interval counter for that endpoint has counted down to zero. This ensures that only one interrupt or isochronous transaction occurs per endpoint per *n* frames (where *n* is the interval set in the USB_TXINTERVAL or USB_RXINTERVAL register for that endpoint).

An active bulk transaction is started immediately, provided there is sufficient time left in the frame to complete the transaction before the next SOF packet is due. If the transaction needs to be retried (for example, because a NAK was received or the target function did not respond) then the transaction is not retried until the transaction scheduler has checked all the other endpoints for active transactions first. This check ensures that an endpoint that is sending a lot of NAKs does not block other transactions on the bus. The USB controller lets you specify a limit (USB_TXINTERVAL or USB_RXINTERVAL registers) to the length of time in which NAKs may be received from a particular target before the endpoint is timed out.

Babble

If the bus is still active at the end of a frame, the USB controller assumes that the function it is connected to has malfunctioned, suspends all transactions, generates a babble interrupt (RESET_OR_BABLE_B), and clears the SESSION bit in the USB_OTG_DEV_CTL register to end the session. This will cause the USB controller to revert to peripheral mode. The USB controller does not start a transaction until the bus is inactive for at least the minimum inter-packet delay. The controller also does not start a transaction unless it can be finished before the end of the frame.

To recover from a babble error condition, the processor must take the following actions inside the interrupt service routine.

1. Turn off VBUS
   Wait until the VBUS level indicator reads b#01.
2. Turn on VBUS
   Wait until the VBUS level indicator reads b#11.
3. Set the SESSION bit

The VBUS level indicator is a field of the USB_OTG_DEV_CTL register.

Because VBUS is sourced external to the processor, make sure that in your hardware design you connect a GPIO to the external source so that you can use software to turn VBUS on and off.

VBUS Events

The USB On-The-Go specification defines a series of thresholds to which the devices involved in point-to-point communications are required to respond.

- VBUS Valid (required to be between 4.4V and 4.75V)
- Session Valid for 'A' device (required to be between 0.8V and 2.1V)
- Session End (required to be between 0.2V and 0.8V)

Which thresholds are critical and the processor response depends upon whether the device is an 'A' device or a 'B' device and the circumstances of the event. These actions are described below.

Actions as an "A" Device

VBUS >VBUS Valid with session initiated by USB controller (that is, VBUS level indicator = b#11 and session bit is set). When VBUS is greater than VBUS Valid, the USB controller selects Host Mode and waits for a device to be connected. It then generates a connect interrupt. The processor resets and enumerates the connected 'B' device.

VBUS > Session Valid with session initiated by 'B' device (that is, VBUS level indicator = b#10 and session bit is clear). When VBUS is greater than Session Valid, the USB controller generates a session request interrupt. The processor sets the session bit and the USB controller either stays in Host mode or changes to Peripheral mode, depending upon the state of the pull-up resistor on the 'B' device. For more information, refer to the Host Negotiation Protocol of the OTG specification. The selected mode is indicated by the state of the Host Mode bit.

VBUS below VBUS Valid while the Session bit remains set (that is, VBUS level indicator ≠ b#11 and session bit is set). This indicates a problem with the VBUS power level. For example, the battery power may have dropped too low to sustain VBUS Valid. Or, the 'B' device may be drawing more current than the 'A' device can provide. In either case, the USB controller will automatically terminate the session and generate a VBUS error interrupt.

To recover from this VBUS error condition, the processor must take the following actions inside the VBUS error interrupt handler.

- Turn off VBUS
  Wait until the VBUS level indicator reads b#01.
- Turn on VBUS
  Wait until the VBUS level indicator reads b#11.
- Set the SESSION bit

The VBUS level indicator is a field of the `USB_OTG_DEV_CTL` register.

Because VBUS is sourced external to the processor, make sure that in your hardware design you connect a GPIO to the external source so that you can use software to turn VBUS on and off.

Actions as a "B" Device

VBUS > Session Valid (that is, VBUS level indicator = b#10 and session bit is clear). This indicates activity from the 'A' device. The USB controller sets the session bit and disconnects the pull down resistor on the D+ line.

VBUS < Session Valid while the session bit remains set (that is, VBUS level indicator = b#01 and session bit is set). This indicates that the 'A' device has lost power (or become disconnected). The USB controller clears the session bit and generates a disconnect interrupt. The processor ends the session.

VBUS < Session End (that is, VBUS level indicator = b#00). This is the condition under which a 'B' device can initiate a session request. If the session bit is set, then after 2ms of SE0 on the bus, the USB controller starts SRP by first pulsing the data line, then pulsing VBUS.

Host Mode Reset

If the RESET bit in the USB_POWER register is set while the USB controller is in host mode, the USB controller generates reset signaling on the bus. The processor core should keep this bit set for 20 ms to ensure correct resetting of the target device. After the processor core has cleared the bit, the USB controller starts its frame counter and transaction scheduler.

Host Mode Suspend

If the SUSPEND_MODE bit in the USB_POWER register is set, the USB controller completes the current transaction then stops the transaction scheduler and frame counter. No further transactions are started and no SOF packets are generated.

To exit suspend mode, the processor core should set the RESUME_MODE bit and clear the SUSPEND_MODE bit in the USB_POWER register. While the RESUME_MODE bit is high, the USB controller generates resume signaling on the bus. After 20 ms, the processor core should clear the RESUME_MODE bit, at which point the frame counter and transaction scheduler are started.

While in suspend mode, the USB controller clock is stopped to reduce power. The SUSPEND_BE output also goes low, if enabled. This feature may be used to power-down the USB drivers. If remote wake-up is to be supported, power to the PHY must be maintained, so the USB controller can detect resume signaling on the bus.

Functional Description

The following sections describe the function of the USB OTG interface.

On-Chip Bus Interfaces

The USB controller uses two independent bus interfaces (peripheral slave and DCB/DEB master) to communicate with a processor-based subsystem. The slave interface allows the processor core to access the control and status registers (including DMA master registers) and the endpoint FIFOs. The master interface is used to drive data into or out of the endpoint FIFOs with minimal processor core interaction.

The peripheral bus slave interface has the following characteristics.

- 16-bit wide transfers
- Wait states are asserted when FIFO accesses take place (maximum of three are possible when contention for the SRAM occurs).

The DCB/DEB bus master interface has the following characteristics:

- 16-bit wide read and write data busses
- write transfers of byte and 16-bit words are possible (byte accesses are used only for remaining bytes in a transfer)
- read transfers of 16 bits (first few or last few bytes may be discarded based on starting address and DMA count respectively)

Interface Pins

The USB OTG external interface has the pins shown in Table 26-2.

Table 26-2. USB 2.0 HS OTG Pins

Signal Name	Input/ Output	Description
USB_DP	I/O	USB D+ pin
USB_DM	I/O	USB D– pin
USB_XI	C	Clock XTAL input 1
USB_XO	C	Clock XTAL input 2
USB_ID	I	USB ID pin
USB_VBUS	I/O	USB VBUS pin
USB_V_{REF}	O	USB voltage reference source (Test purposes only)
USB_RSET	O	USB resistance set (Test purposes only)

Power and Clocking

The USB controller uses the system clock `CLK` (greater than 30 MHz required) to generate an internal clock used to clock the USB registers. The transceiver clock is a 60MHz clock sourced from the UTMI PHY and is used by the PHY interface logic and USB engine. The A 32 KHz clock (refer to the `USB_SRP_CLKDIV` register) is used for D+ pulse detection for SRP signaling by an OTG 'B' device only.

During SUSPEND and when no session is active, the clock to much of the USB controller is stopped to reduce power consumption. The clock becomes operational again when RESUME signaling is detected on the USB lines.

UTMI Interface

The interface to the on-chip PHY uses the industry-standard UTMI+ (universal transceiver macro interface) level 2. This provides full high-speed device and OTG functionality, but does not support communication to a hub.

The PHY is a mixed-signal block and includes the following:

- full-speed and high-speed drivers and receivers (single-ended and differential)
- data line pull-up and pull-down resistors
- full-speed and high-speed CDR
- VBUS and `USB_ID` level detection
- host disconnect detection
- full-speed/high-speed shift registers, NRZI encode/decode and bit-stuff encode/decode

Although the UTMI specification indicates that VBUS charging, driving and discharging be done inside the PHY, for process-restricting and power reasons, these functions are typically implemented off-chip in a separate USB charge-pump chip.

Programming Model

The following sections describe the USB OTG programming model.

Peripheral Mode Flow Charts

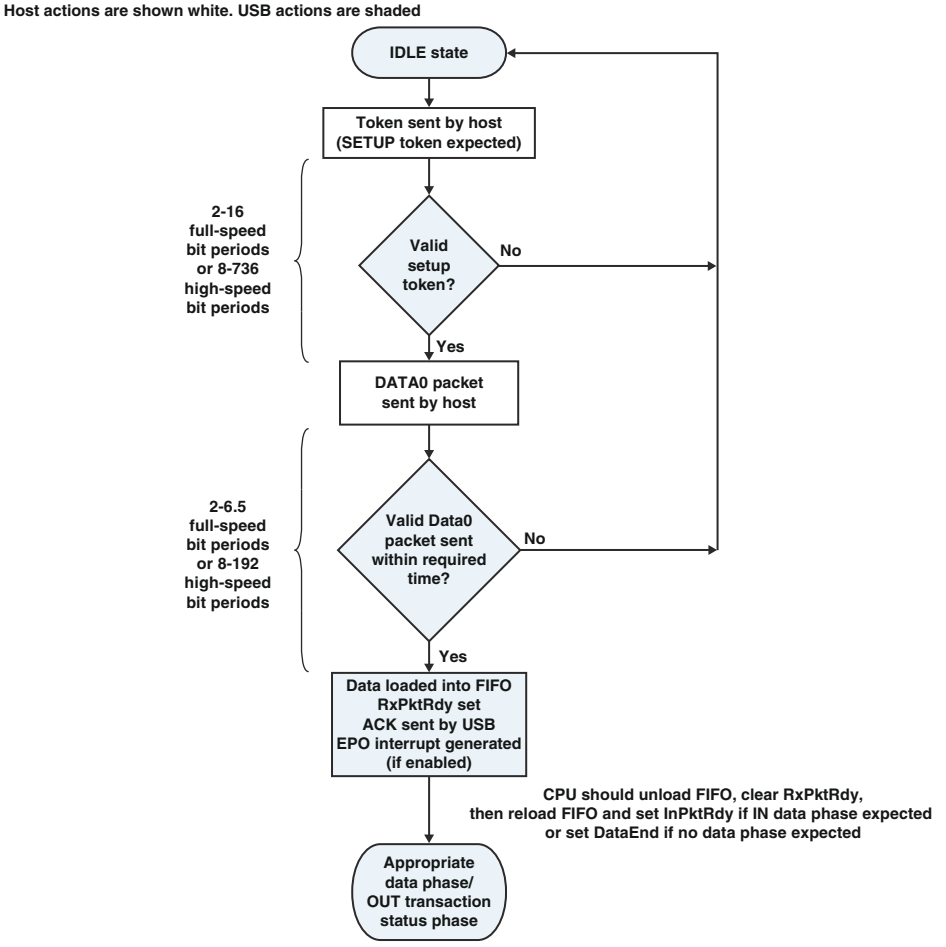


Figure 26-8. USB Control Setup Phase

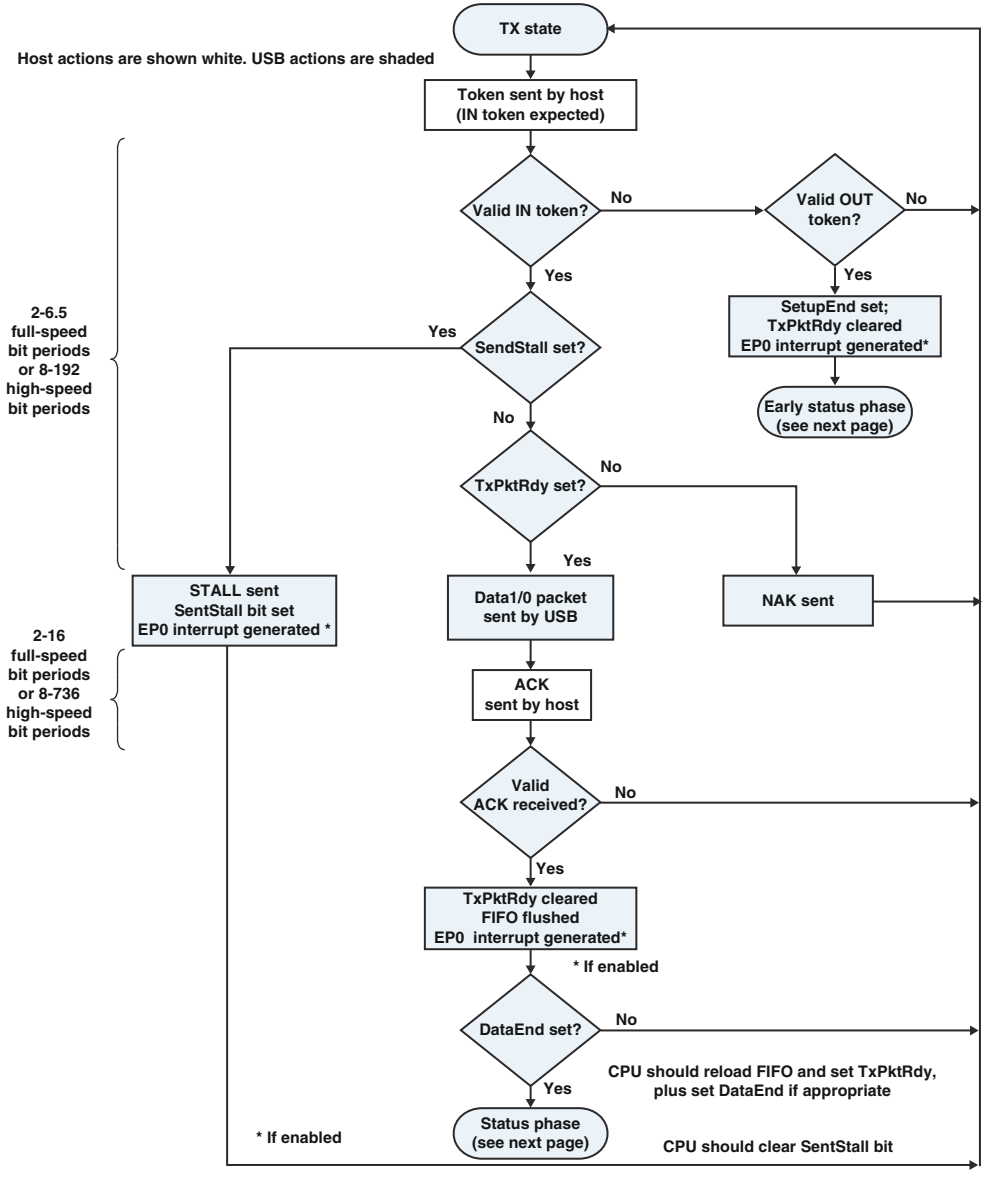


Figure 26-9. Control In Data Phase

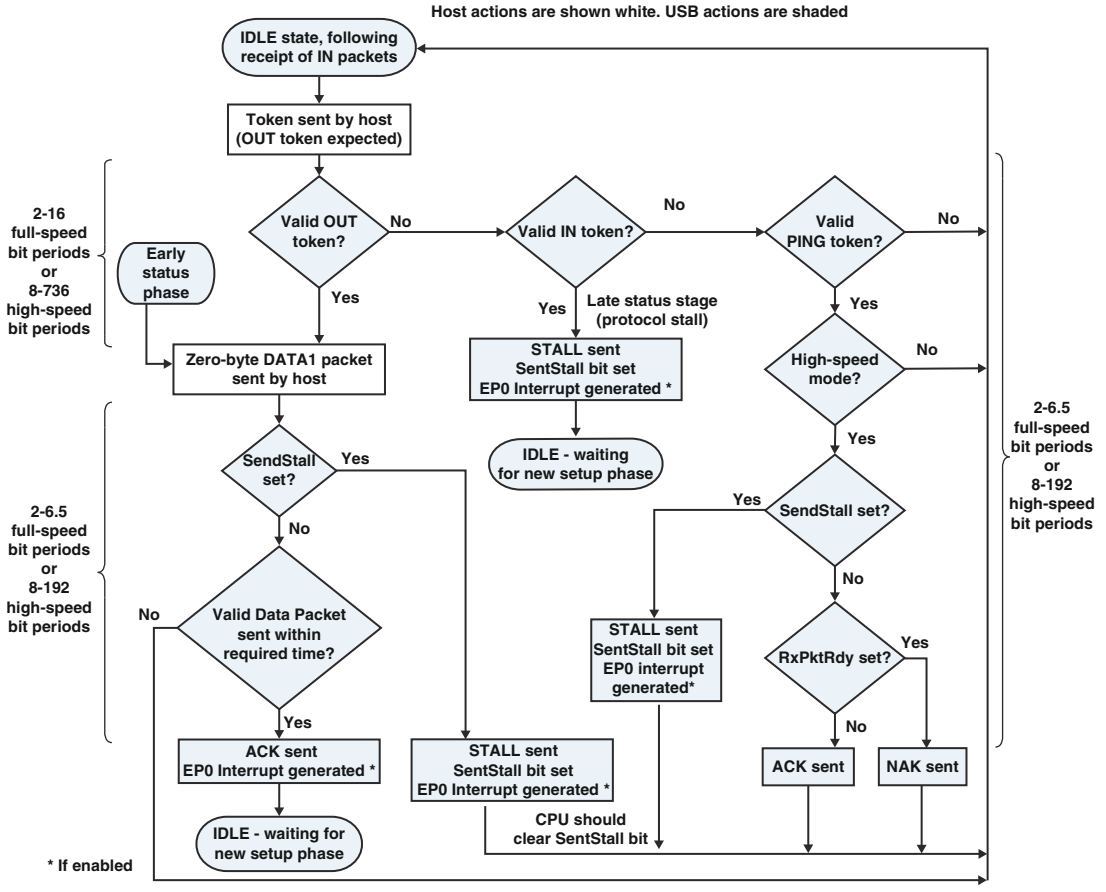


Figure 26-10. Control In Data Status Phase

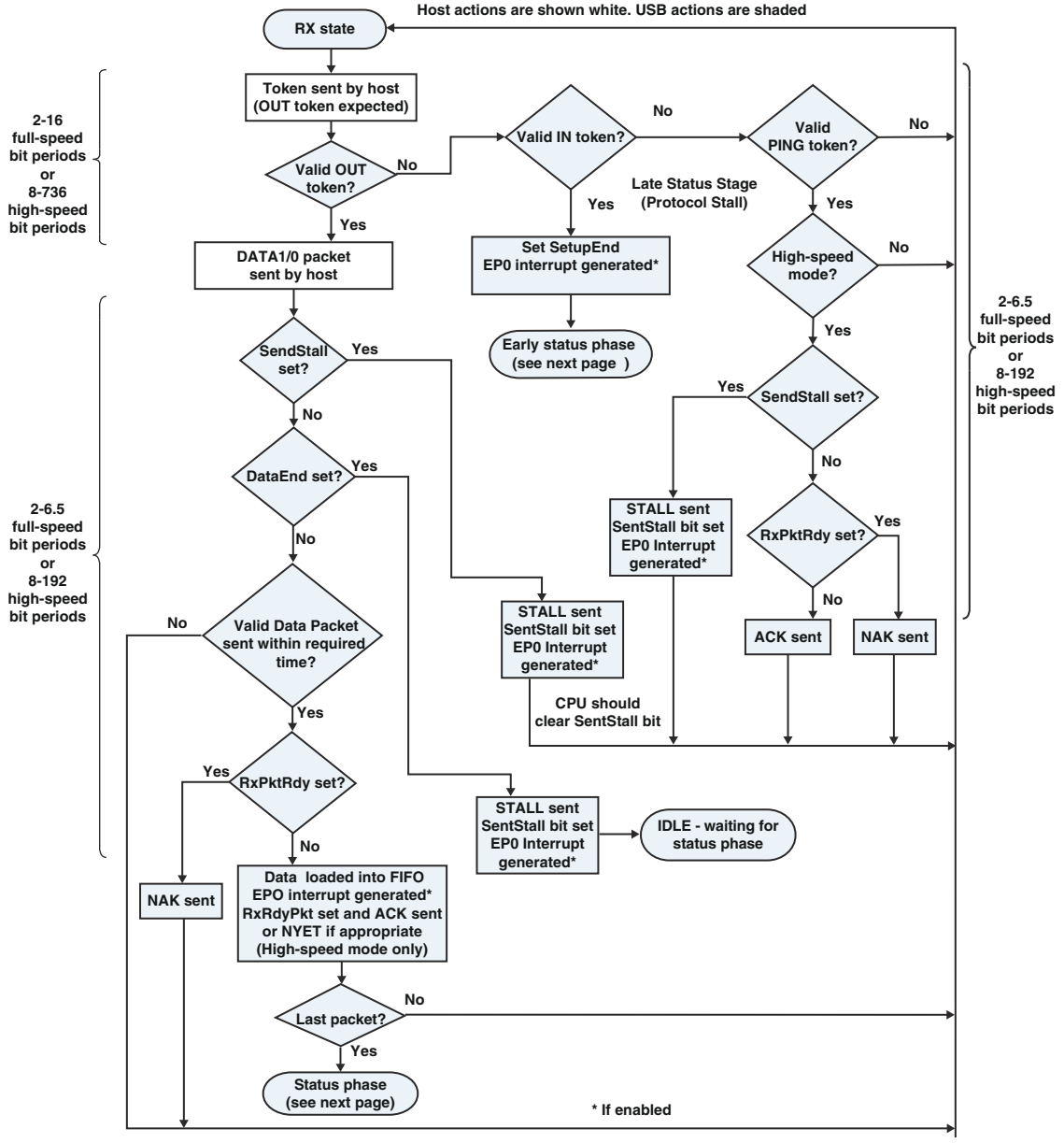


Figure 26-11. Control Out Data Phase

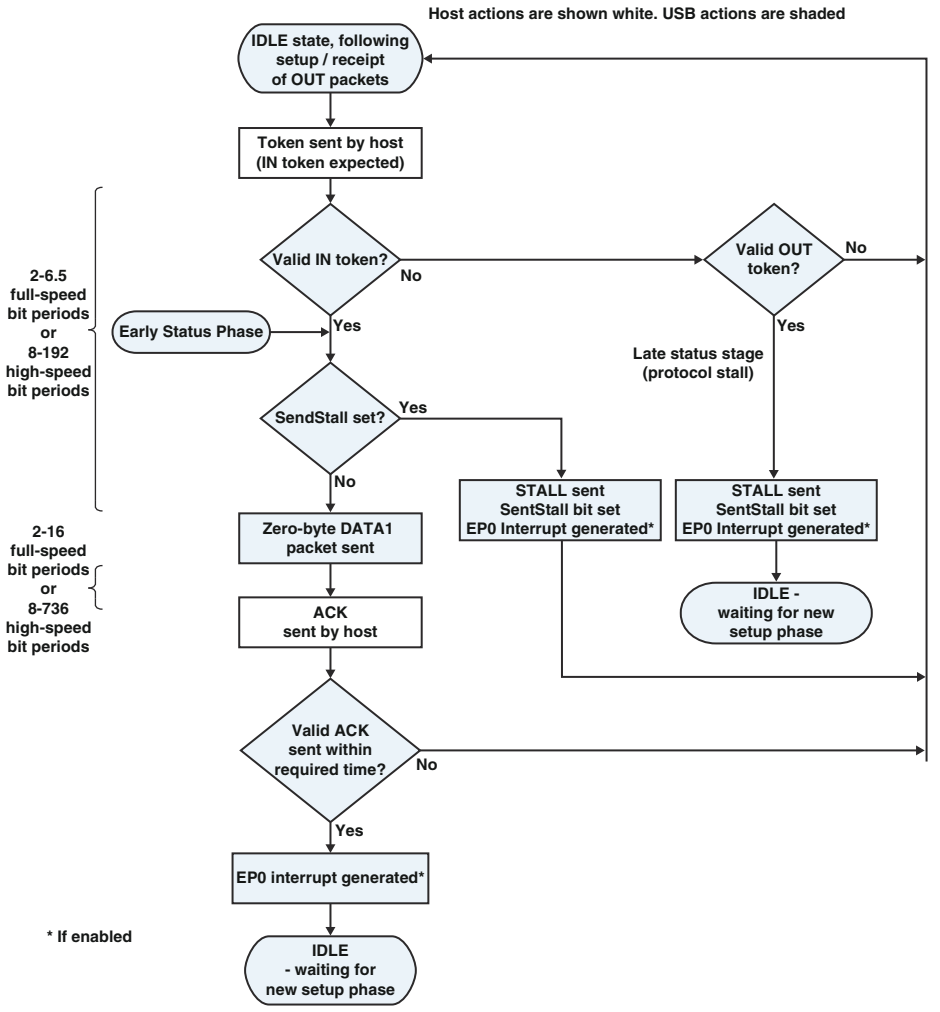


Figure 26-12. Control Out Data Status Phase

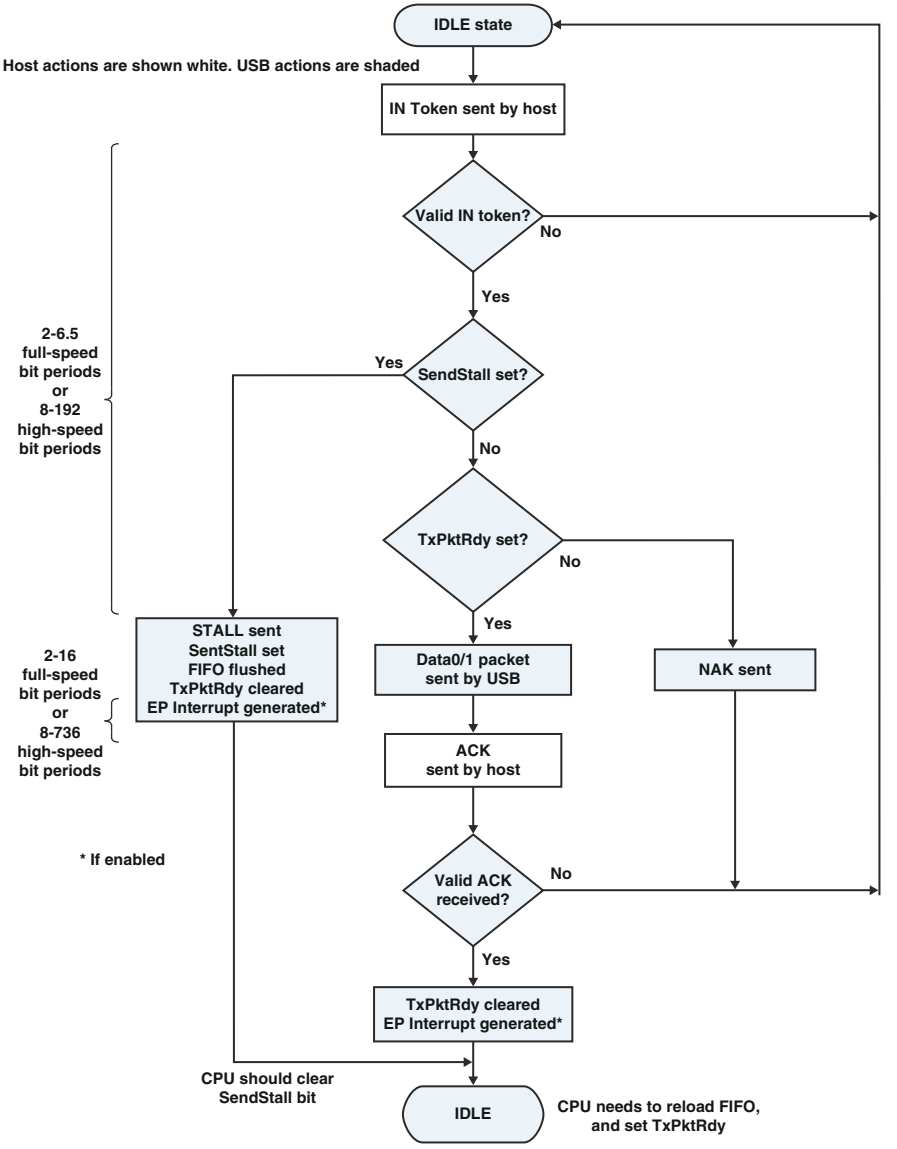


Figure 26-13. Bulk/Low Bandwidth Interrupt In Transaction

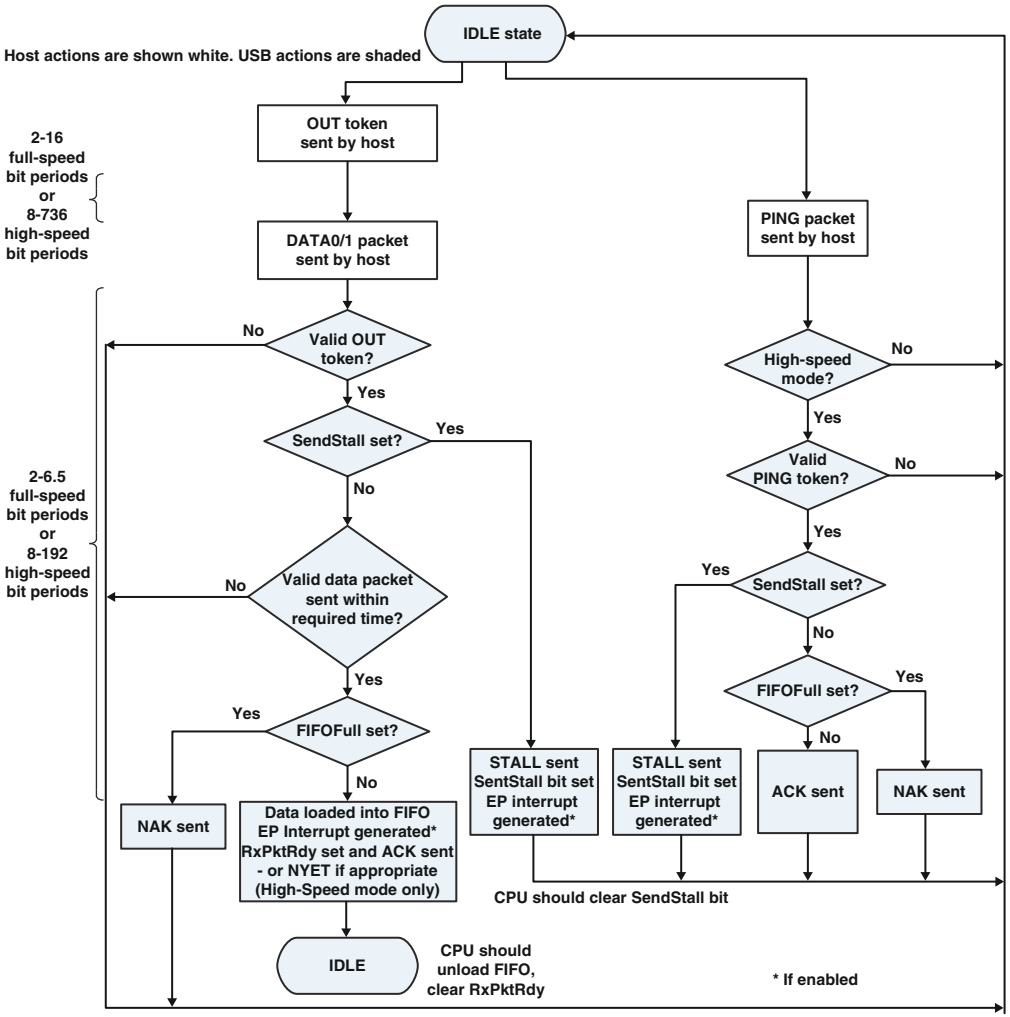


Figure 26-14. Bulk/Low Bandwidth Interrupt Out Transaction

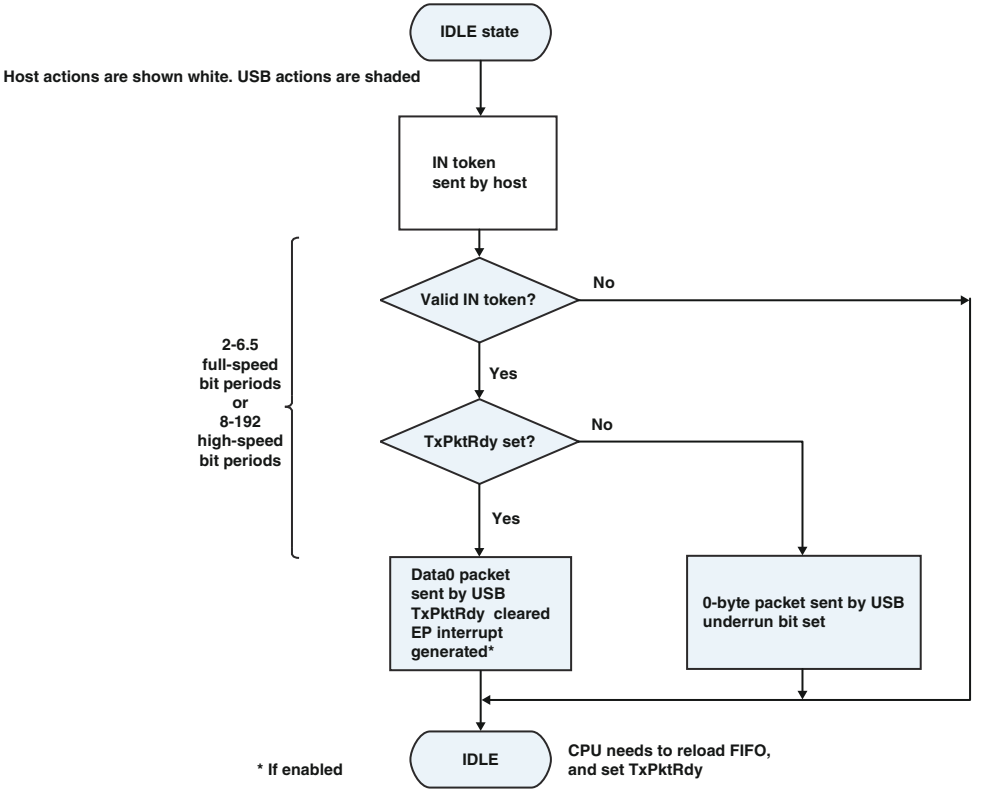


Figure 26-15. Full-speed/Low Bandwidth Isochronous In Transaction

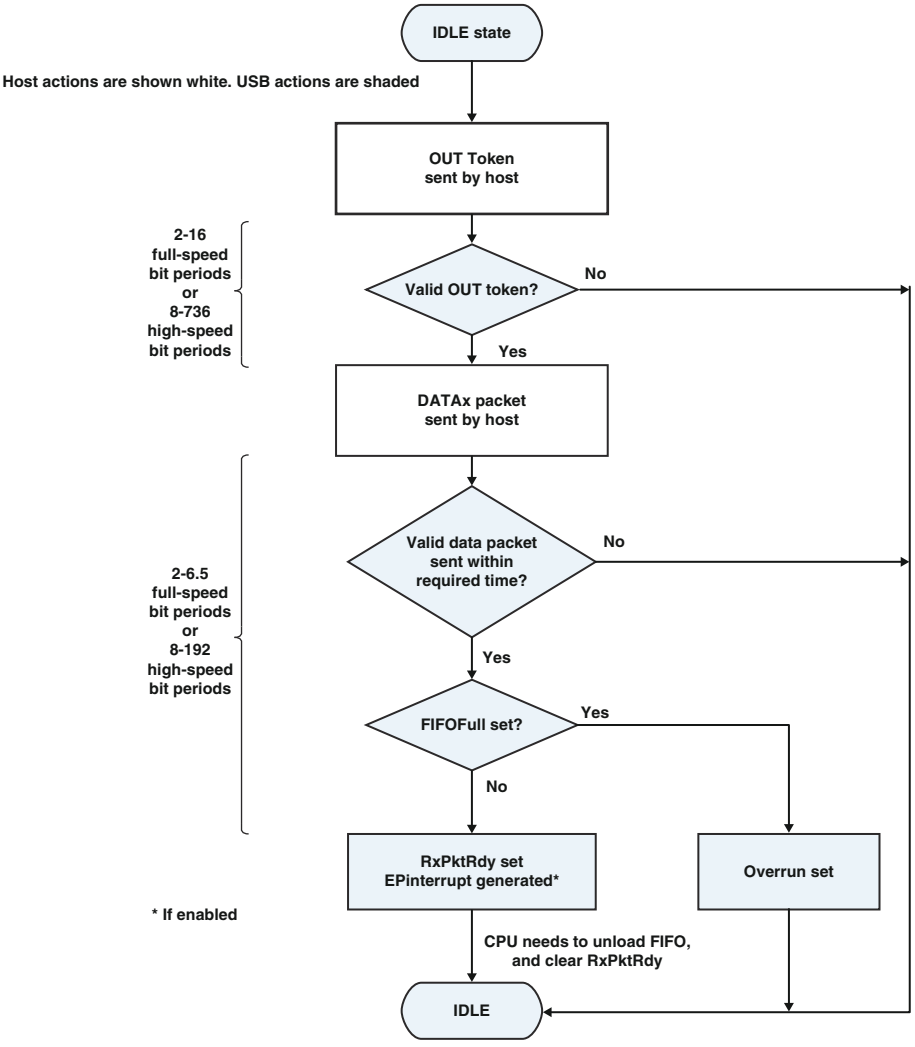


Figure 26-16. Full-speed/Low Bandwidth Isochronous Out Transaction

Host Mode Flow Charts

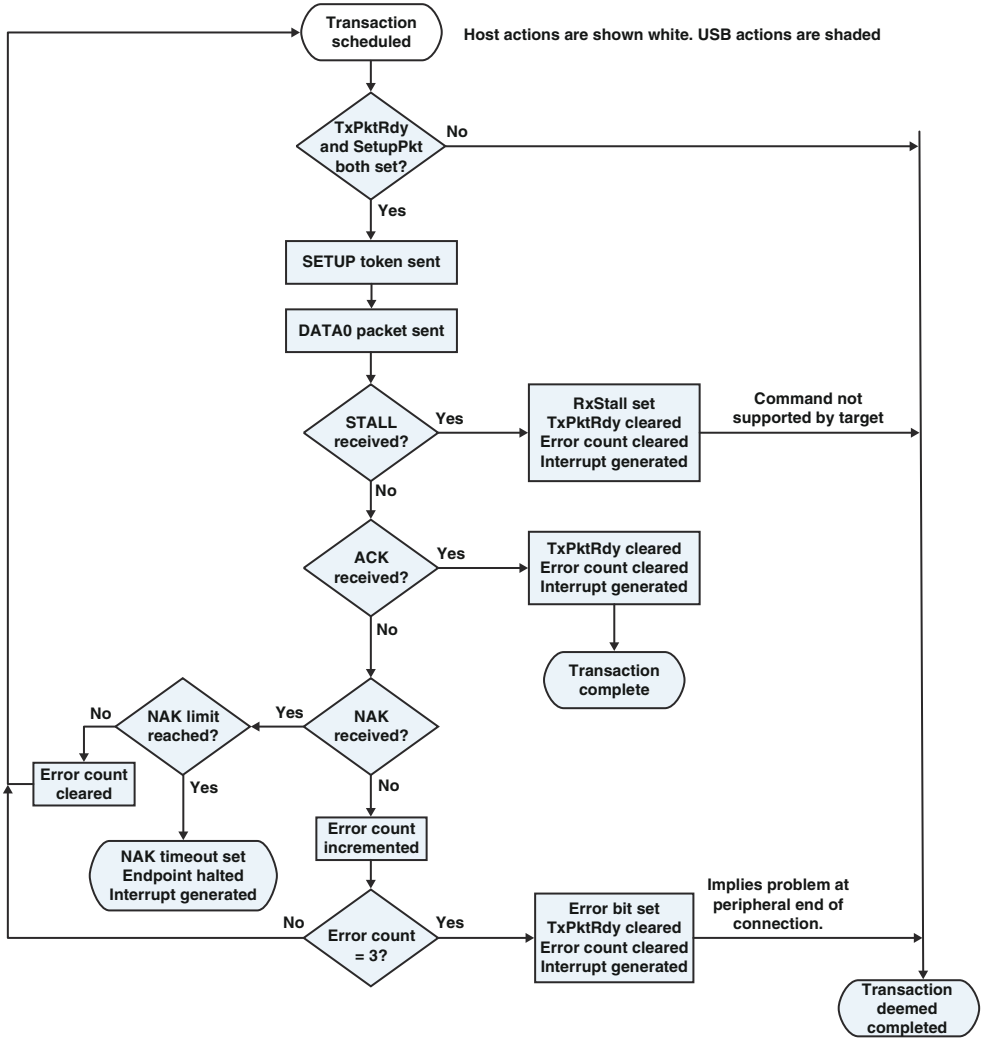


Figure 26-17. USB Control Setup Phase

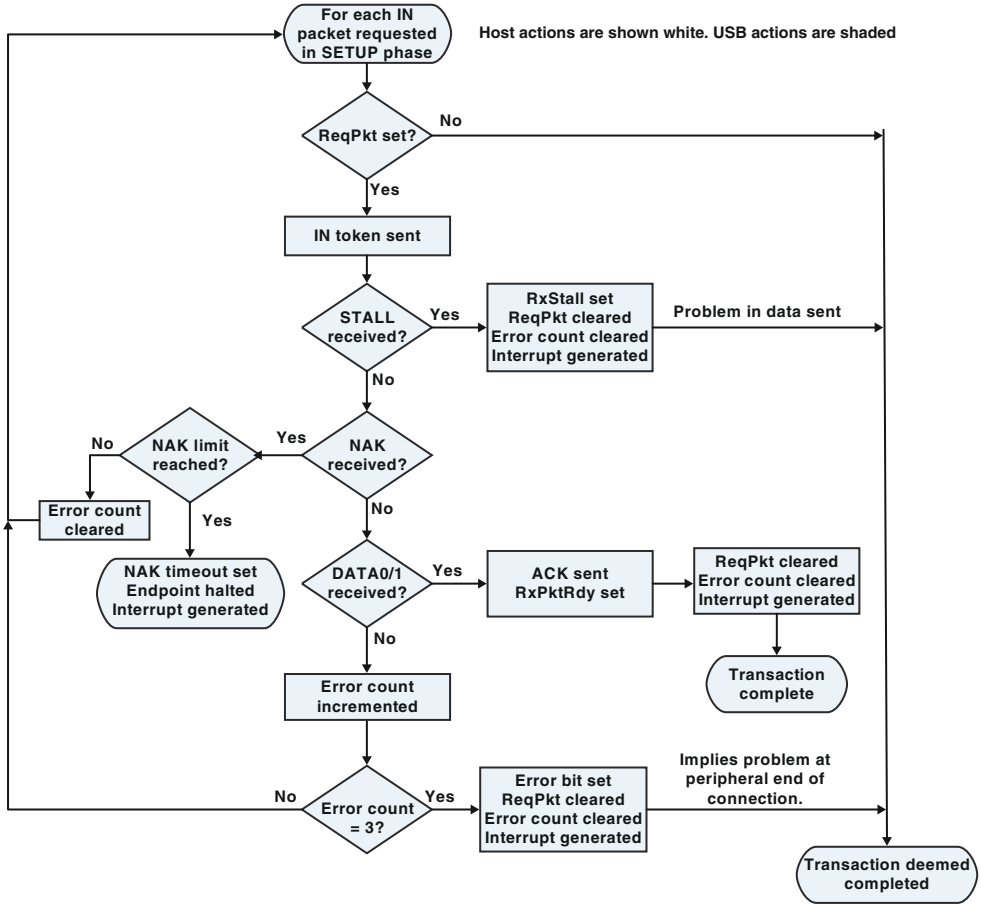


Figure 26-18. Control In Data Phase

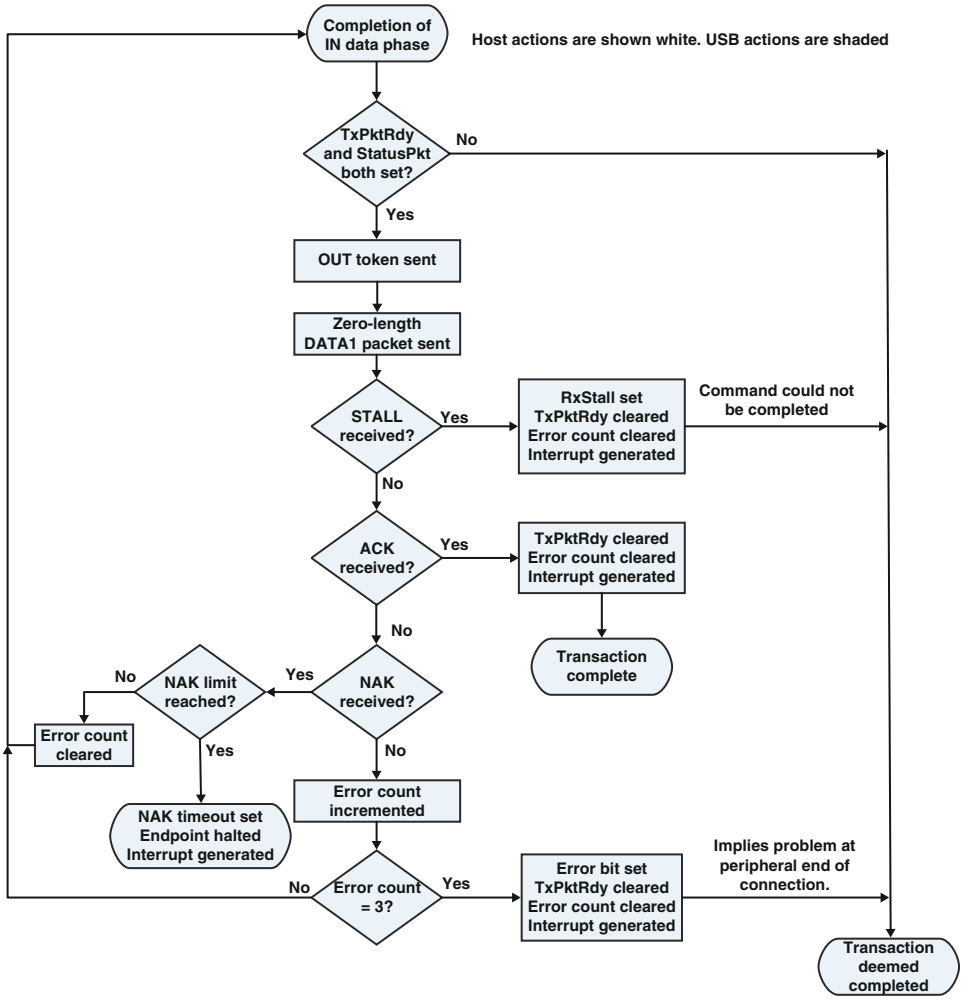


Figure 26-19. Control In Data Status Phase

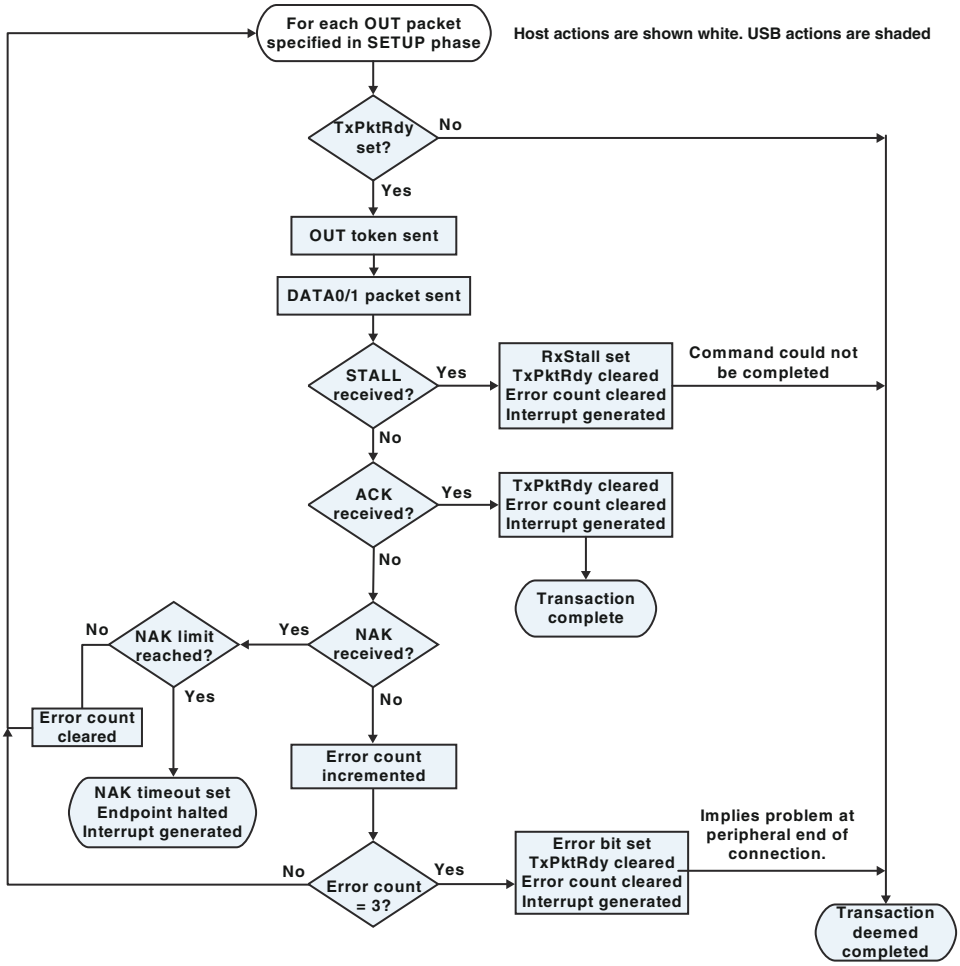


Figure 26-20. Control Out Data Phase

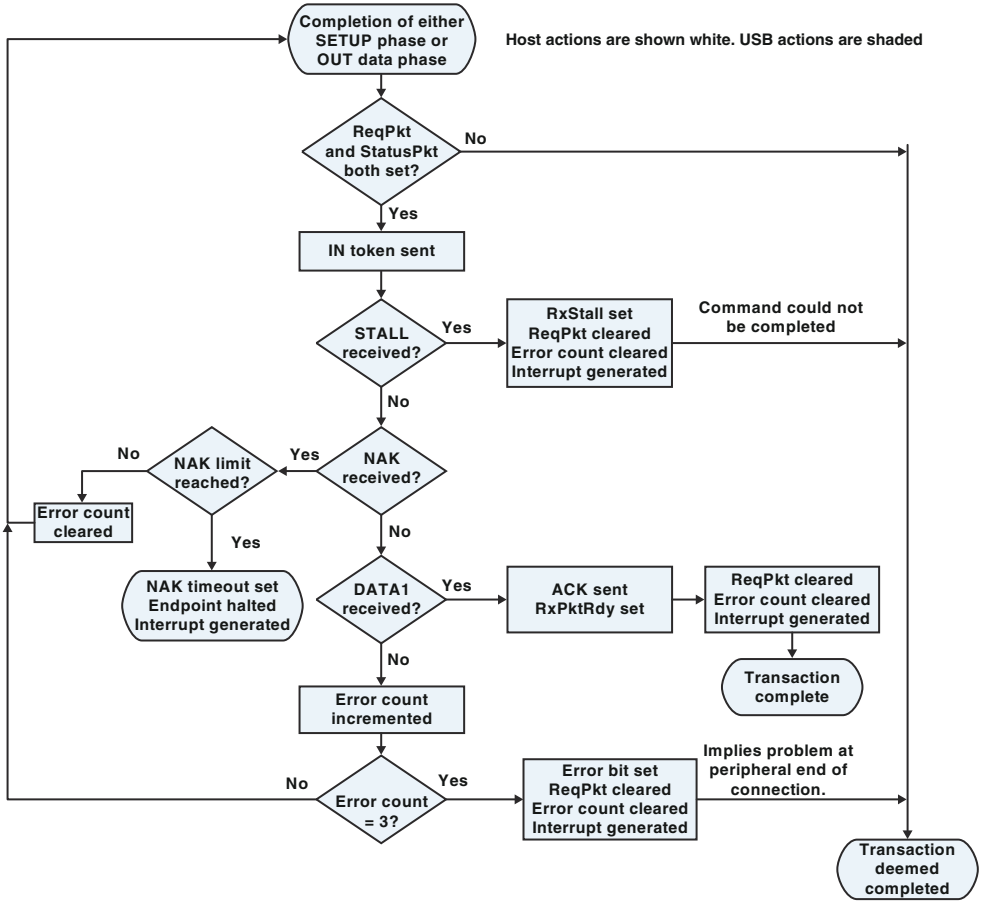


Figure 26-21. Control Out Data Status Phase

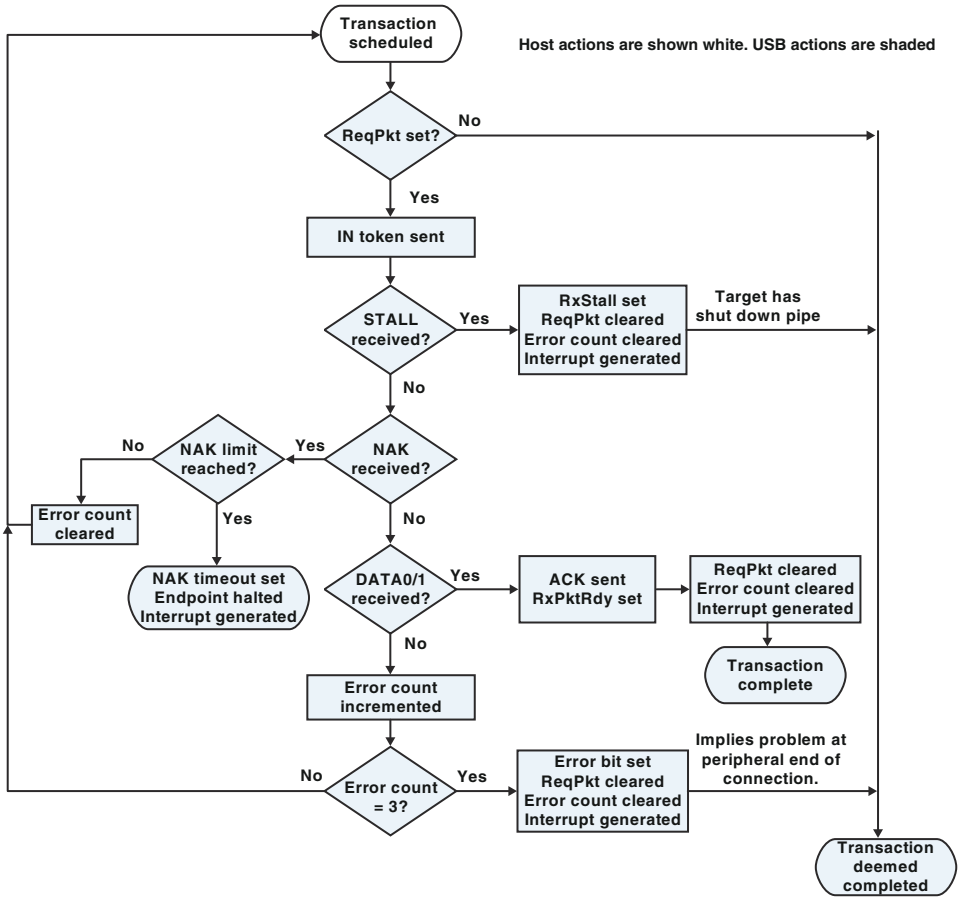


Figure 26-22. Bulk/Low Bandwidth Interrupt In Transaction

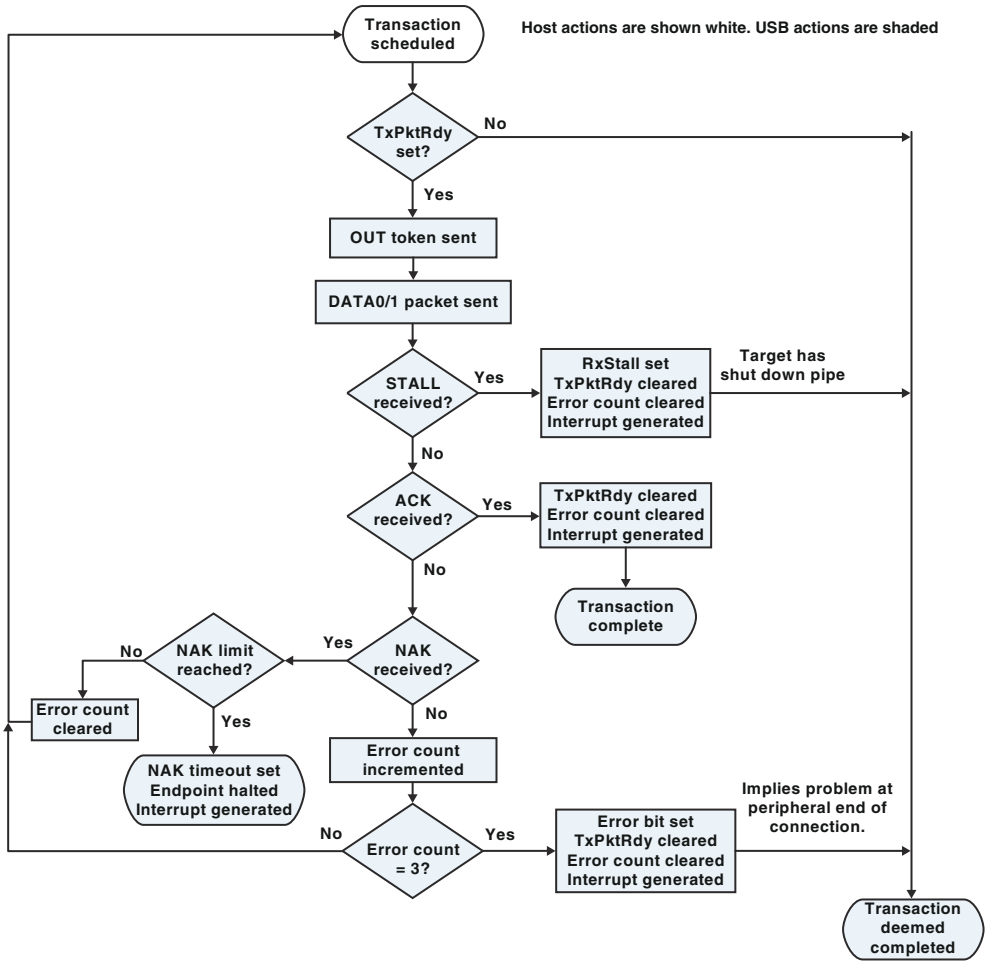


Figure 26-23. Bulk/Low Bandwidth Interrupt Out Transaction

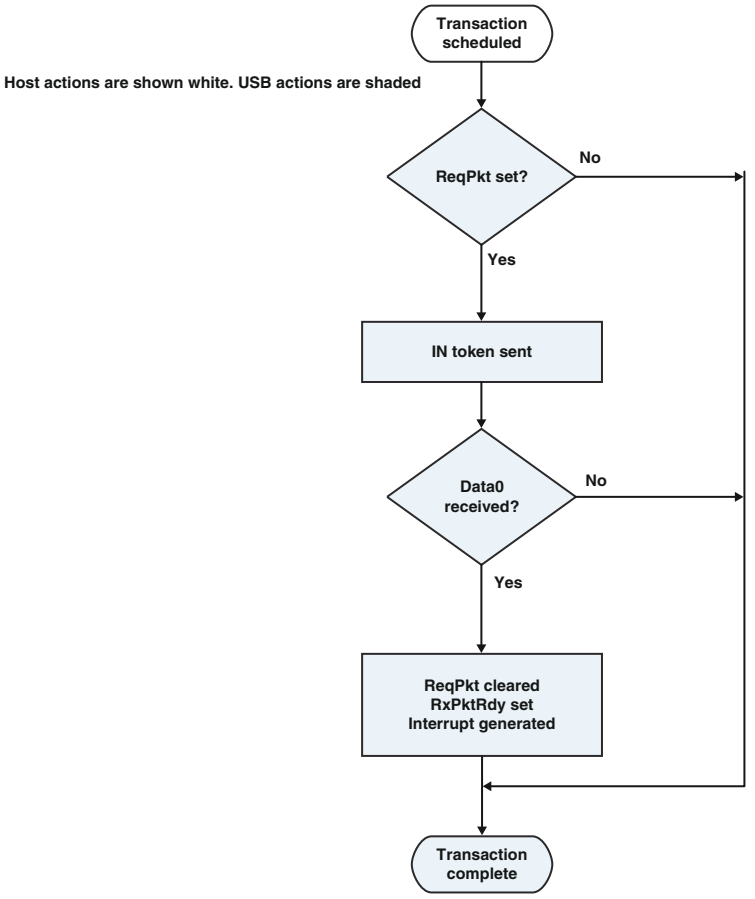


Figure 26-24. Full-speed/Low Bandwidth Isochronous In Transaction

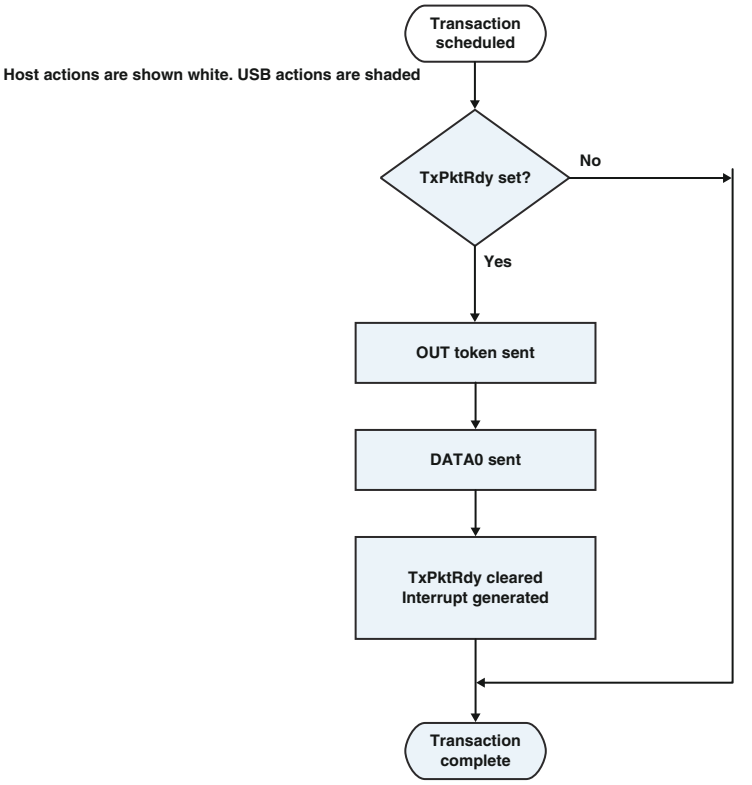


Figure 26-25. Full-speed/Low Bandwidth Isochronous Out Transaction

DMA Mode Flow Charts

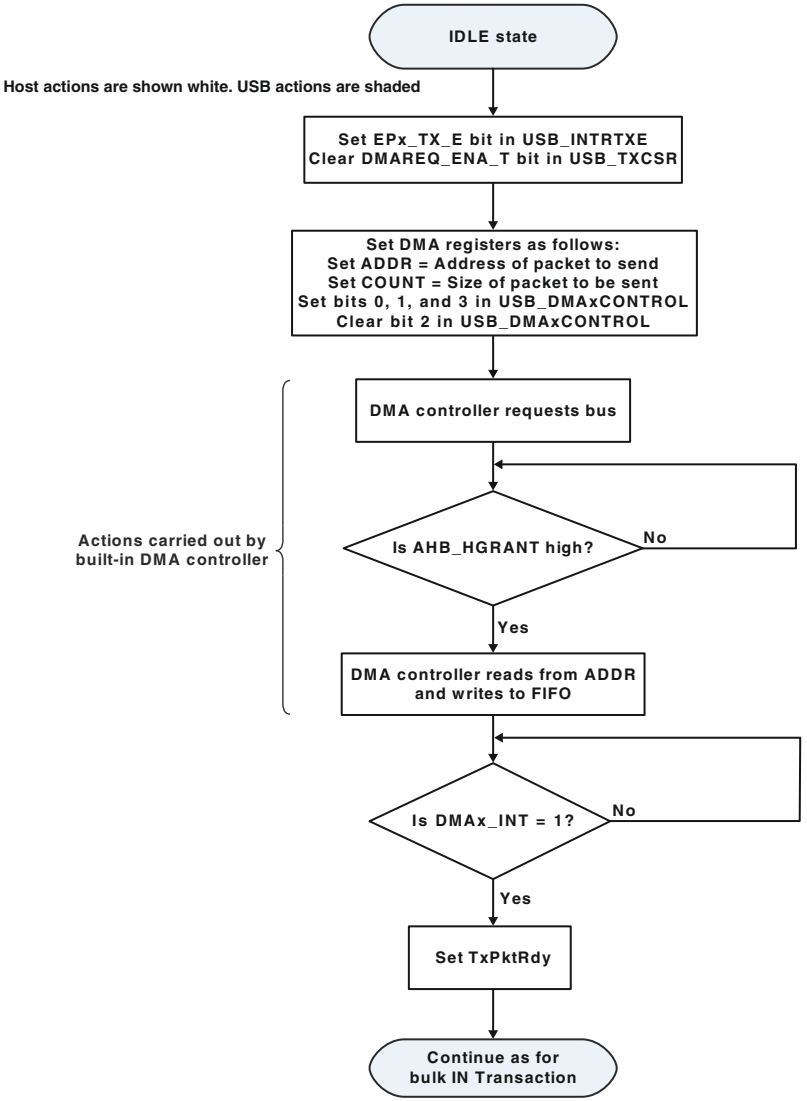


Figure 26-26. Single Packet Transmit During DMA Operation

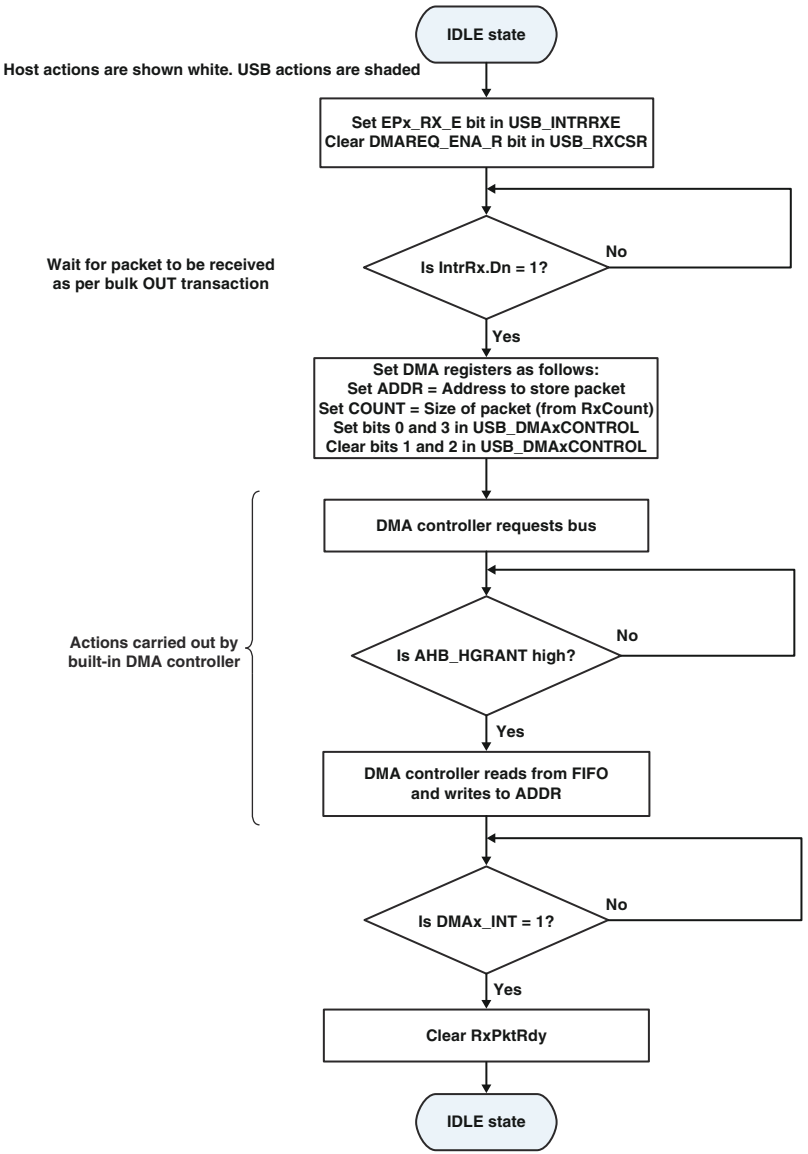


Figure 26-27. Single Packet Receive During DMA Operation

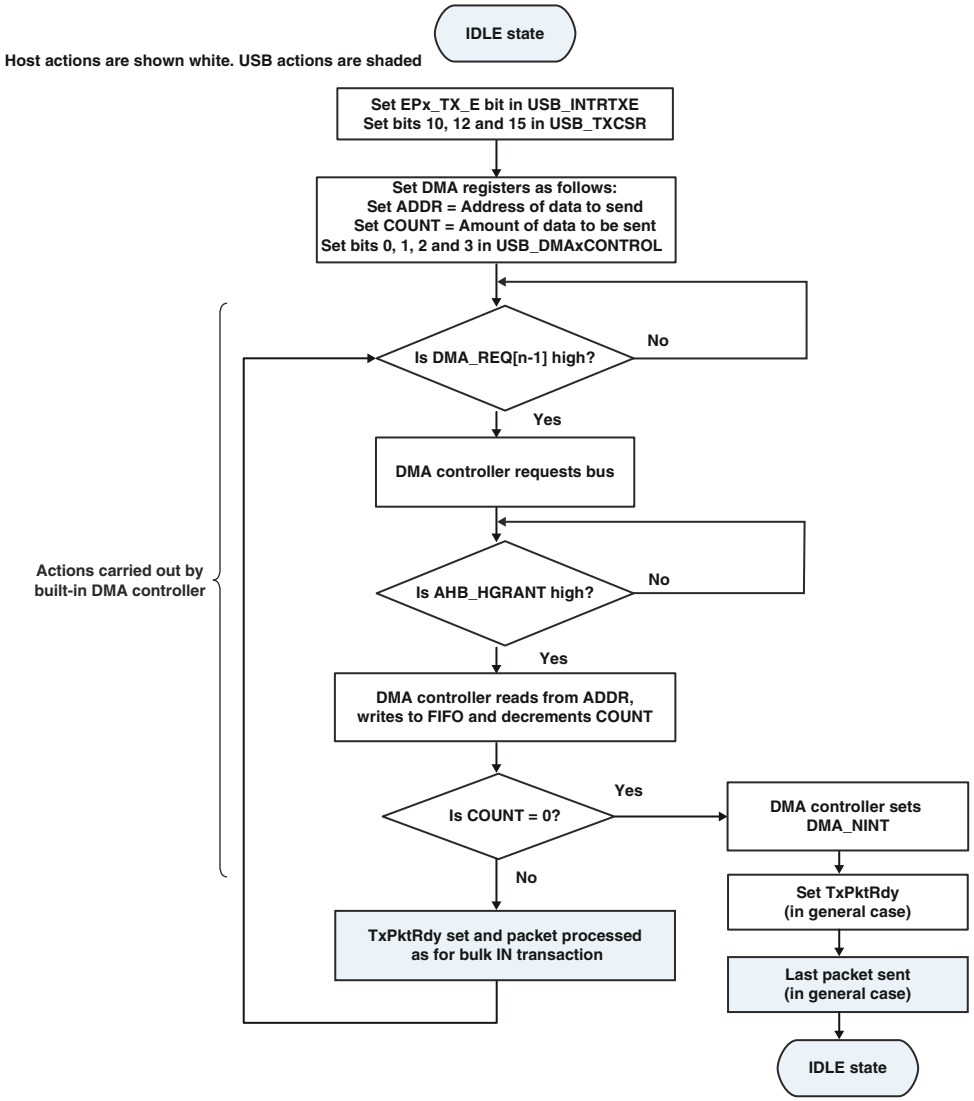


Figure 26-28. Multiple Packet Transmit During DMA Operation

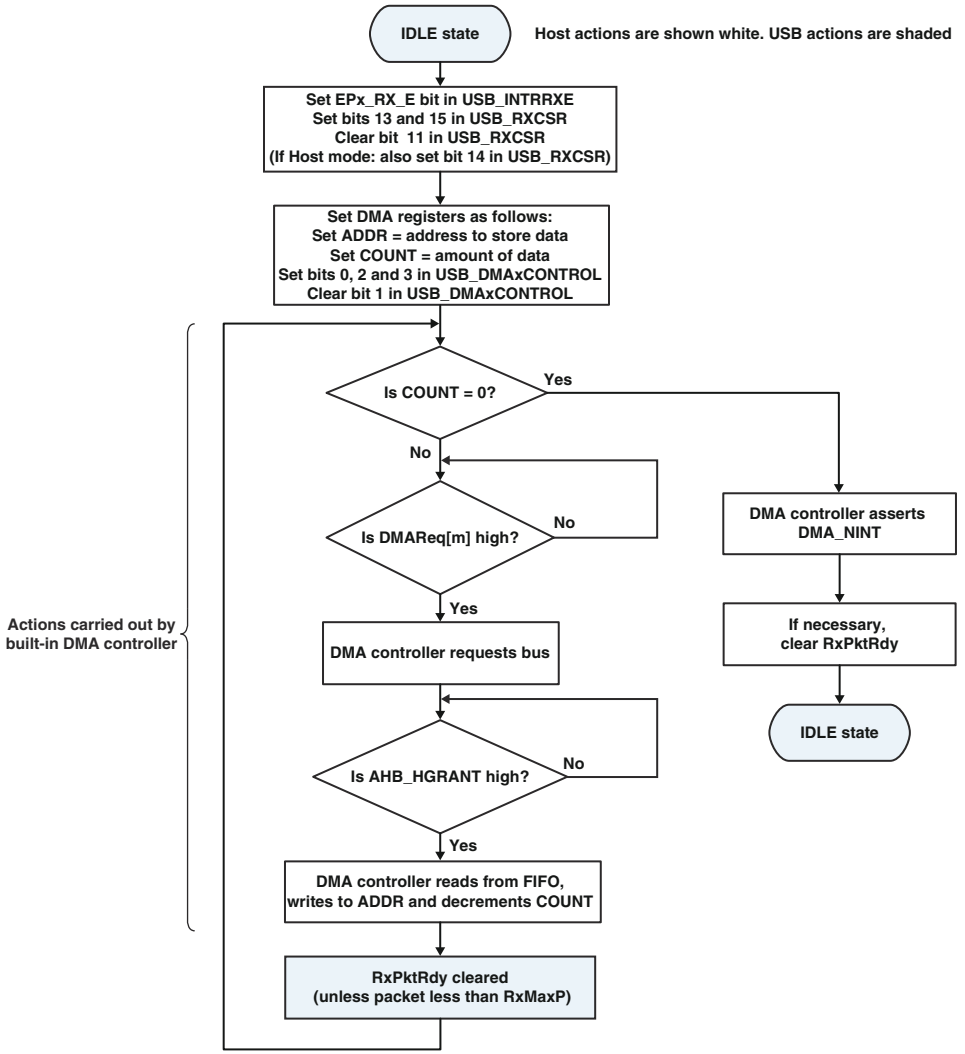


Figure 26-29. Multiple Packet Receive During DMA Operation (Data Size Known)

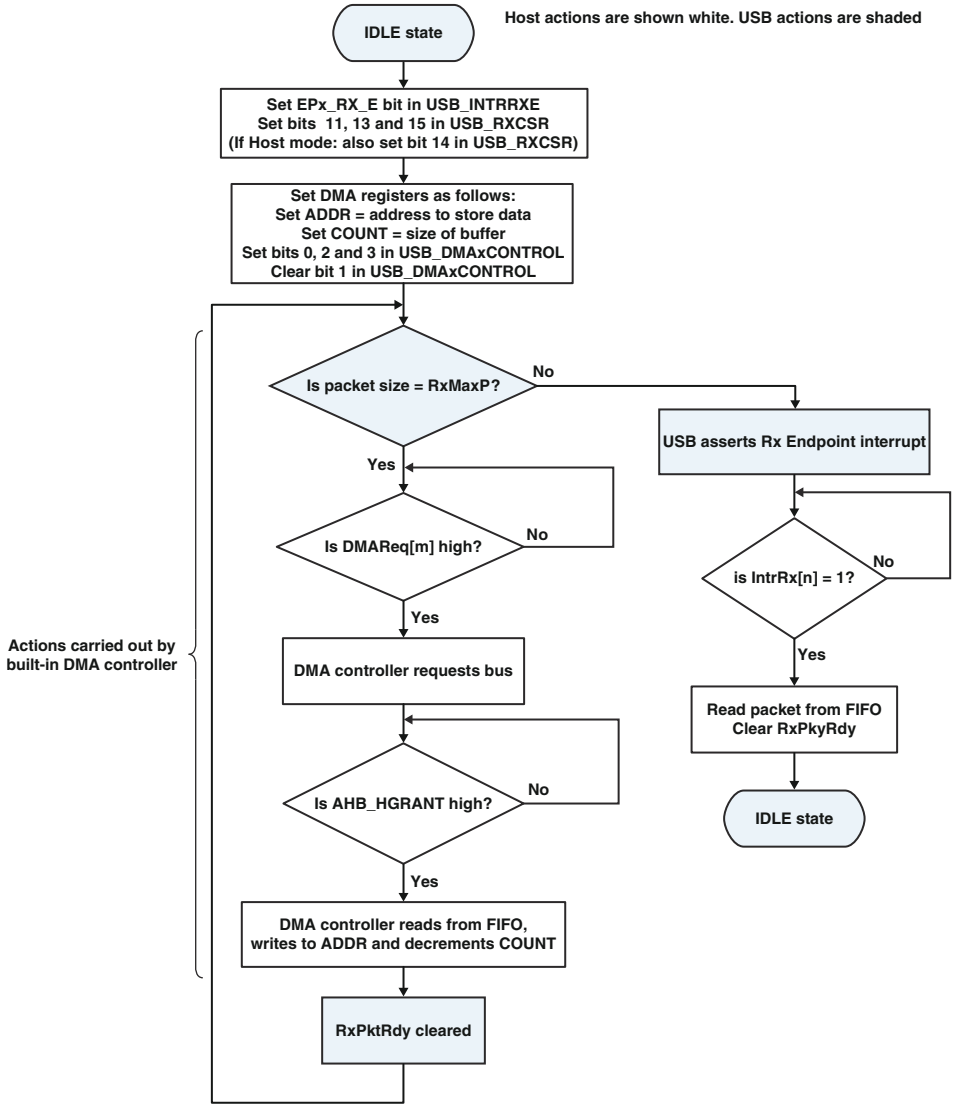


Figure 26-30. Multiple Packet Receive During DMA Operation (Data Size Not-known)

OTG Session Request

In order to conserve power, the USB on-the-go supplement allows VBUS to only be powered up when required and to be turned off when the bus is not in use.

VBUS is always supplied by the 'A' device on the bus. The USB controller determines whether it is the 'A' device or the 'B' device by sampling the USB_ID input from the PHY. This signal is pulled low when an A-type plug is sensed (signifying that the USB controller is the 'A' device), but the input is taken high when a B-type plug is sensed (signifying that the USB controller is the 'B' device).

Starting a Session

When the device containing the USB controller wants to start a session, the processor core must set the SESSION bit in the USB_OTG_DEV_CTL register. The USB controller then enables ID pin sensing. This results in the USB_ID input either being taken low if an A-type connection is detected or high if a B-type connection is detected. The B_DEVICE bit in the USB_OTG_DEV_CTL register is also set to indicate whether the USB controller has adopted the role of the 'A' device or the 'B' device.

If the USB controller is the 'A' device: The USB controller then enters host mode (the 'A' device is always the default host), and waits for VBUS to go above the VBUS valid threshold, as indicated when the VBUS1-0 bits in the USB_OTG_DEV_CTL register go to 11.

The Blackfin USB controller does not source VBUS, except when initiating SRP. As such, VBUS must be provided by an external regulator or USB charge pump. The external VBUS supply must be able to be switched off and on. This is required so that the USB controller can recover from VBUS errors or Babble conditions.

The USB controller then waits for a peripheral to be connected. When a peripheral is detected, a connect interrupt (CONN_B bit in USB_INTRUSB) is generated (if enabled) and either the FSDEV or LSDEV bit in the USB_OTG_DEV_CTL register is set, depending on whether a full-speed peripheral or a low-speed peripheral was detected. The processor core should then reset this peripheral. To end the session, the processor core should clear the SESSION bit in USB_OTG_DEV_CTL.

If the USB controller is the 'B' device: The USB controller requests a session using the session request protocol defined in the USB on-the-go supplement (for example, it first asserts the DISCHRG_VBUS_START bit in USB_OTG_VBUS_IRQ to discharge VBUS). Then, when VBUS has gone below the session end threshold (as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 00), and the line state is SE0 for greater than 2 ms, the USB controller first pulses the data line then pulses VBUS (by taking the interrupt CHRG_VBUS_START in USB_OTG_VBUS_IRQ high).

At the end of the session, the SESSION bit is cleared – usually by the USB controller but it can also be cleared by the processor core if the application software wishes to perform a software disconnect. For more information, see the description of "USB OTG Device Control (USB_OTG_DEV_CTL) Register" on page 26-134. The USB controller switches on the pull-up resistor on D+. This signals to the 'A' device to end the session.

Detecting Activity

When the other device of the OTG set-up wants to start a session, it *either* raises VBUS above the session valid threshold (if it is the 'A' device as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 10), *or (*if it is the 'B' device) first pulses the data line then pulses VBUS. Depending on which of these actions happens, the USB controller can determine whether it is the 'A' device or the 'B' device in the current set-up and act accordingly.

If VBUS is raised above the session valid threshold, the USB controller is the 'B' device. The USB controller sets the SESSION bit in the USB_OTG_DEV_CTL register. When reset signaling is detected on the bus, a reset interrupt (RESET_OR_BABLE_B =1) is generated (if enabled) that the processor core should interpret as the start of a session. The USB controller is in peripheral mode at this point as the 'B' device is the default peripheral.

At the end of the session, the 'A' device turns off the power to VBUS. When VBUS drops below the session valid threshold (as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 01), the USB controller detects this and clears the SESSION bit to indicate that the session has ended. A disconnect interrupt (DISCON_B bit in USB_INTRUSB) is also generated (if enabled).

If data line/VBUS pulsing is detected, the USB controller is the 'A' device. The controller generates a SESSION_REQ_B interrupt (bit 6 in USB_INTRUSB, if enabled) to indicate that the 'B' device is requesting a session. The processor core should then start a session by setting the SESSION bit.

Host Negotiation/Configuration

When the USB controller is the 'A' device (USB_ID low, B_DEVICE= 0), the controller automatically enters host mode when a session starts.

When the USB controller is the 'B' device (USB_ID high, B_DEVICE= 1), the controller automatically enters peripheral mode when a session starts. The processor core can request that the USB controller become the host by setting the HOST_REQ bit in the USB_OTG_DEV_CTL register. This bit can be set either when requesting a session start by setting the SESSION bit in USB_OTG_DEV_CTL or at any time after a session has started. When the USB controller next enters suspend mode (no activity on the bus for 3 ms), and assuming the HOST_REQ bit remains set, the controller enters host mode and begins host negotiation (as specified in the USB OTG supplement), causing the PHY to disconnect the pull-up resistor on the D+ line. This

should cause the 'A' device to switch to peripheral mode and to connect its own pull-up resistor. When the USB controller detects this, it generates a connect interrupt (CONN_B bit in USB_INTRUSB) if this is enabled. The controller also sets the RESET bit in the USB_POWER register to begin resetting the 'A' device. (The USB controller begins this reset sequence automatically to ensure that reset is started as required within 1 ms of the 'A' device connecting its pull-up resistor). The processor core should wait at least 20 ms, then clear the RESET bit and enumerate the 'A' device.

When the USB controller-based 'B' device has finished using the bus, the processor core should put it into suspend mode by setting the SUSPEND_MODE bit in the USB_POWER register. The 'A' device should detect this and either terminate the session or revert to host mode. If the 'A' device is USB controller-based, it generates a disconnect interrupt (DISCON_B bit in USB_INTRUSB) if this is enabled.

Software Clock Control

Power consumption is minimized in the USB controller by software-controlled clock propagation. The USB_GLOBAL_CTL register is used to enable clocks to only those parts of the controller that are necessary to perform a given USB function. The GLOBAL_ENA bit must be set in order to do any operations with the USB, even including writing to other registers. Endpoint 0 control and FIFO access depends on the GLOBAL_ENA bit.

The remaining endpoint 1 – 7 TX and RX register access, transfer operation and FIFO access is dependent on the corresponding bit of USB_GLOBAL_CTL being set. State is retained in the registers when the particular endpoint clock is stopped.

Wakeup from Hibernate State

To conserve power when the chip is idle, systems often uses powerdown modes to shut down power and clocks to various parts of the chip. Hibernate state saves the most power (core clock, peripherals clocks, and internal power are off; only external power is on).

During the course of normal operation, the software can decide that the chip has been idle for a long enough period that there is no immediate need for the clocks to be active and the chip can be put into a power-down mode such as hibernate. This period of inactivity occurs when there is a USB suspend state (idle on the bus for greater than 3 ms) or if no OTG session is valid. The SUSPEND_MODE bit (in USB_POWER) and VBUS1-0 status bits (in USB_OTG_DEV_CTL) are used to indicate these states.

Before the system software (driver) pushes processor into the hibernate state, the software has to make sure that the CSR_HBR bit (in USB_APHY_CNTRL2) is set. Setting this bit activates the non-idle activity detection logic in the PHY. Any non-idle activity on the USB bus is detected by the non-idle activity detection logic in the analog PHY. This logic wakes up the processor and generates a low to high transition on EXT_WAKE pin.

To be able to use non-idle activity detection logic as a wakeup source for the processor, enable the USB wakeup source by programming the appropriate bits in the voltage regulator control register (VR_CTL). After the processor wakes up, USB is listed as the wakeup source in the PLL status (PLL_STAT) register. The EXT_WAKE pin can be used by the external power-up sequence chip to power up SDRAM or an other external peripheral. The processor typically goes through these steps (see Figure 26-31 on page 26-85) when it comes out of hibernate state.

After the chip comes out of hibernate state, the software has to make sure that the CSR_RSTD bit of the USB_APHY_CNTRL2 register is set. This setting deactivates the non-idle activity detection logic and ensures proper USB functionality.

The interrupt will be asserted when either of the following events occur:

- Non-idle signaling occurs during the USB suspend state (including USB reset signaling)
- VBUS falls below the session valid threshold

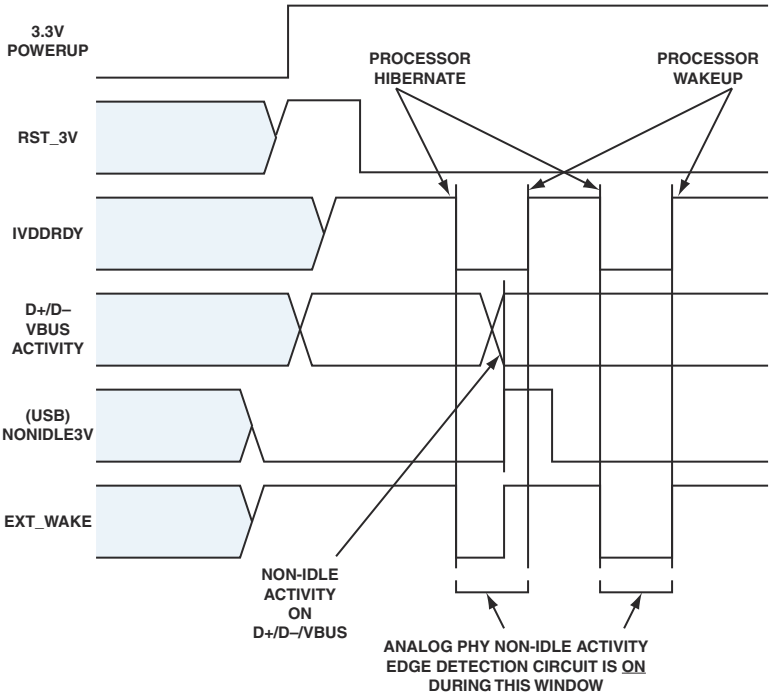


Figure 26-31. Timing Diagram of EXT_WAKE Pin

Wakeup Without Re-Enumeration

When USB goes into suspend mode after 3 ms of inactivity on the D+ and D–, it is possible that the processor is pushed into the hibernate state. Hibernate state implies that all internal power is shut down, and only the external 3.3 V power is present. And, all the clocks in the processor are shut down. If the USB were to wake up in response to non-idle activity on

the D+ and D–, the USB controller would have lost the state it was in before going to hibernate. This lost state would cause the host to re-enumerate USB controller device. To prevent re-enumeration of the USB device, system software must do the following.

- Before the system software (driver) pushes the processor into hibernate, it must make sure that the state of the USB is stored in external memory flash.
- Also, the software must make sure that the CSR_HBR bit is set in the USB_APHY_CNTRL2 register.

A low to high transition on CSR_HBR generates a pulse (high) on the csr hbr 1v signal (internal USB controller signal). This signal is used by the USB analog PHY to retain the states of the pull-up and pull-down resistors during the hibernate state. Retaining the states of the pull-up and pull-down resistors on D+ and D– implies to the host that the USB controller device is not disconnected from the USB bus.

After the system software pushes the processor into hibernate state, any non-idle activity on the USB bus is detected by the non-idle activity detection logic in the analog PHY. After the processor wakes up from the hibernate state, the processor typically goes through these steps: powering up the processor, waiting for the PLL to lock, and booting the code into L1 memory.

After code is loaded into L1 memory, it is executed. The executed code restores the state of the USB to pre-hibernate state. After the state is resumed, the analog PHY no longer needs to retain the state of the pull-ups and pull-downs on D+ and D–. The system software has to make sure that CSR_RSTD bit is set in the USB_APHY_CNTRL2 register. A low to high transition on the CSR_RSTD bit generates a pulse on the csr_rstd 1v signal (internal USB controller signal). This signal is used by the analog PHY to prevent holding the values of pull-up and pull-down resistors. The

pull-ups and pull-downs are now controlled by the USB controller. This sequence of actions (see Figure 26-32) prevents re-enumeration of the USB controller device after the processor wakes up from hibernate state.

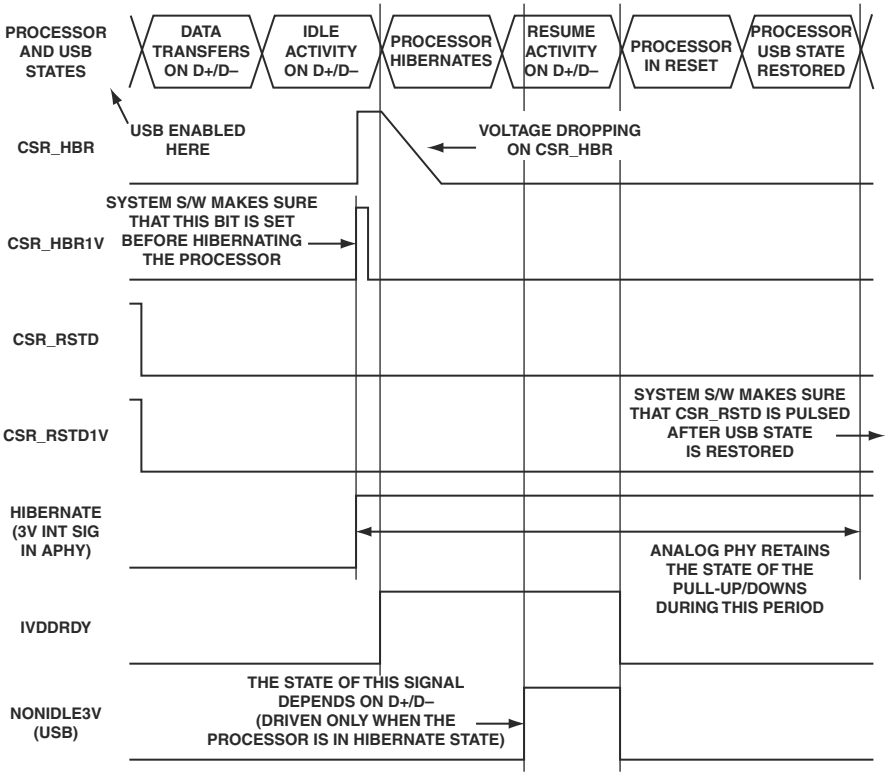


Figure 26-32. Timing Diagram of the CSR_HBR/CSR_RSTD Bits

Data Transfer

Regardless of whether the USB controller is operating in host or peripheral mode, data is channeled through the endpoint FIFOs to construct packets to be sent or to be received over the USB. The RX FIFOs are used to receive OUT packets when in peripheral mode and IN packets when operating in host mode. Similarly, the TX FIFOs are used to transmit IN packets when in peripheral mode and OUT packets as a host.

Data may be moved between the FIFOs and memory using either DMA or interrupts. Each endpoint FIFO has its own individually programmable options so that each can be set up separately. Different transfer types must be treated differently by the system. Data transfers of significant size almost certainly require DMA to move the data around; but smaller packet sizes might be handled completely by the processor.

Each data endpoint supports both double and single-buffering modes. In single-buffered operation, FIFOs are unloaded and loaded on a packet-by-packet basis. Double-buffering imposes less burden on the system by allowing two packets to be buffered in a FIFO before it is necessary to use DMA/interrupts to service the FIFO. Double-buffering mode is automatically enabled when a *MaxPktSize* is set for an endpoint that is equal to or less than half the size in bytes of that FIFO.

Loading/Unloading Packets from Endpoints

Because the peripheral bus slave interface to the USB controller provides a fixed transfer size of half words (16-bits), some additional work is required to use packet or transfer sizes that are an odd-number of bytes in length. This prevents data loss or corruption. This situation only exists for FIFO interface accesses through the processor core slave interface (DMA mastered endpoints can access individual bytes).

For TX endpoints with an odd number of bytes to be written into the FIFO, there is the possibility that an extra byte could be incorrectly written. The USB controller provides hardware counting and comparison logic to prevent this from occurring. When writing such a packet into the USB controller, the following steps are required.

- Load the appropriate `USB_TXCOUNT` register with the packet/transfer size in bytes.
- Write all the data into the FIFO (using DMA or processor core) with the final half word of the transfer containing the final byte aligned to the least significant byte lane.

After a `USB_TXCOUNT` register is loaded with a value, it counts down the number of bytes written into that particular FIFO on each processor core or DMA write. When there is only one byte remaining in the transfer, the USB controller latches the least significant byte of the last half word.

Another use for the `USB_TXCOUNT` registers is to streamline DMA transfers, preventing unnecessary processor interaction in lengthy multi-packet transfers.

For RX endpoints using odd packet/transfer sizes, the software must compensate for the fact that the least significant byte lane of the final half word in the transfer is valid.

For EP0 RX transfers, if the last packet is not a multiple of four bytes it is strongly recommended that the remainder (n bytes mod 4) be unloaded from the FIFO using a special byte addressing FIFO register (EP0 FIFO address + 4). This prevents the USB controller from sending non-null data during the status phase of the control transfer.

DMA Master Channels

The USB controller provides eight DMA master channels to provide a more efficient transfer of larger amounts of data between the FIFOs and the processor core; and to free up the processor core for other tasks. Each of these channels is configured and controlled using the DMA control registers.

Each DMA controller can operate in one of two DMA modes: 0 or 1. When operating in mode 0, the DMA controller only can be programmed to load or unload one packet, so processor intervention is required for each packet transferred over the USB. This mode can be used with any endpoint, whether it uses control, bulk, isochronous, or interrupt transactions.

When operating in DMA mode 1, the DMA controller can only be programmed to load/unload a complete bulk transfer, which can be many packets. After set up, the DMA controller loads or unloads the packets, interrupting the processor only when the transfer has completed. DMA mode 1 can only be used with endpoints that use bulk transactions. DMA mode 1 is most valuable where large blocks of data are transferred to a bulk endpoint. The USB protocol requires such packets to be split into a series of packets of *MaxPktSize* for the endpoint. Mode 1 can be used to avoid the overhead of having to interrupt the processor after each individual packet; instead the processor is only interrupted after the transfer has completed. In some cases, the block of data transferred comprises a pre-defined number of these packets that the controlling software counts through the transfer process. In other cases, the last packet in the series may be less than the maximum packet size and the receiver may use this “short” packet to signal the end of the transfer. If the total size of the transfer is an exact multiple of the maximum packet size, the transmitting software should send a null packet for the receiver to detect.

Each channel can be independently programmed for the selected operating mode.

For bulk OUT transfers using DMA mode 1, the DMA request line is asserted only when there is an edge transition of the state of the RXPKTRDY and a payload of MaxPacketSize has been received. If a data packet has been sitting in the FIFO prior to setting DMAREQMODE1 in USB_RXCSR, the DMA request line will not be asserted when the DMA is enabled in the DMAx_Control register. This will cause the data not to be read from the RX FIFO, resulting in a DMA "hang". However, since the packet arrived before DMAREQMODE and DMAREQ_ENA were enabled in USB_RXCSR, an RX interrupt will be generated for the corresponding endpoint. Therefore, the software should set the DMAREQMODE to Request Mode 0 to unload the pre-received packet. The RX interrupt service routine may look something like this:

Figure 26-33. EP RX Interrupt Service Routine

```
If USB_RXCOUNT == MaxPktSize
Switch to DMA Mode 0 and unload the packet
(in Mode 0, DMA_REQ is always asserted whenever there's data in the FIFO)
You should set the DMA_COUNT to MaxPktSize so as to unload only one packet
If AUTOCLEAR is set, you do not need to manually clear RXPKTRDY
Switch back to DMA Mode 1 and set the count to
(Total_Count – MaxPktSize)
Else
Handle as normal for case of short packet
```

DMA transfers may be 8-bit or 16-bit. All the transfer associated with one packet (with the exception of the last) must be of the same width, so that the data is consistently byte-aligned or word-aligned. The last transfer may contain fewer bytes than the previous transfers in order to complete an odd-byte or odd-word transfer.

DMA Bus Cycles

The DMA controller uses incrementing bursts of an unspecified length on the peripheral DMA bus. The controller starts a new burst when it is first granted bus mastership (whether at the start of a USB packet or when regaining the bus after being thrown off part way through a packet) and when the peripheral address starts a new 1K byte block.

When unloading packets from the FIFOs, the DMA controller requests ahead to the USB controller. Although it starts the transfer with two BUSY cycles while it is getting the first word from the FIFO, all subsequent words of the packet are immediately available. No further BUSY cycles are required. The DMA controller is associated with a two-word buffer, so no data is lost if it loses bus mastership part way through unloading a packet. When bus mastership is regained, it can continue unloading the packet without adding any BUSY cycles.

The DMA start address (written to the `DMAxADDR`) must be word aligned.

Split transactions and retries are supported.

Transferring Packets Using DMA

Use of the DMA master channels to access the USB controller FIFOs requires that both the appropriate channel and the endpoint be programmed appropriately. Many variations are possible. The following sections detail the standard setups used for the basic actions of transferring individual packets and multiple packets.

Individual Packet: RX Endpoint

The transfer of individual packets is normally carried out using DMA mode 0. The USB controller RX endpoint is programmed as follows:.

1. The relevant EPx_RX_E bit in the USB_INTRRXE register is set to 1.
2. The DMA_ENA bit of the appropriate USB_RXCSR register is set to 0. (There is no need to set the USB controller to support DMA for this operation.)
3. When a packet is received by the USB controller, it generates the appropriate endpoint interrupt (using USB_INTRRX). The processor should then program the appropriate DMA master channel as follows:
    - DMAxADDR: memory address to store packet
    - USB_DMAxCOUNT: size of packet (determined by reading the USB controller USB_RXCOUNT register)
    - USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 0, DMAREQMODE_R = 0

The DMA controller then requests bus mastership and transfers the packet to memory. It interrupts the processor when it has completed the transfer. The processor should then clear the RXPKTRDY bit in the USB_RXCSR register.

Individual Packet: TX Endpoint

Again using DMA mode 0, a USB controller TX endpoint is programmed as follows.

1. The relevant EPx_TX_E bit in the USB_INTRTXE register is set to 1.
2. The DMA_ENA bit of the appropriate USB_TxCSR register is set to 0. (There is no need to set the USB controller to support DMA for this operation.)
3. When the FIFO can accommodate data, the USB controller interrupts the processor with the appropriate TX endpoint interrupt. The processor should then program the DMA channel as follows:
    - DMAxADDR: memory address of packet to send
    - USB_DMAxCOUNT: size of packet to be sent
    - USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 1, DMAREQMODE_T = 0

The DMA controller then requests bus mastership and transfers the packet to the USB controller FIFO. When it has completed the transfer, it generates a DMA interrupt. The processor should then set the TXPKTRDY bit in the USB_TXCSR register.

Multiple Packets: RX Endpoint

Multiple packets normally are transferred using DMA mode 1. The DMA controller is programmed using the DMA registers:

- DMAxADDR: memory address of the buffer in which to store transfer
- USB_DMAxCOUNT: maximum size of data buffer
- USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 0, DMAREQMODE_R = 1

The USB controller RX endpoint should now be programmed as follows:

1. The relevant EPx_RX_E bit in the USB_INTRRXE register is set to 1.
2. The AUTOCLEAR_R, DMAREQ_ENA_R and DMAREQMODE_R bits of the appropriate USB_RXCSR register is set to 1. In host mode, the AUTOREQ_RH and DMAREQMODE_RH bits should also be set to 1.

As each packet is received by the USB controller, the DMA master channel requests bus mastership and transfers the packet to memory. With AUTOCLEAR_R set, the USB controller automatically clears its RXPKTRDY bit. This process continues automatically until the USB controller receives a short packet (one of less than the maximum packet size for the endpoint) signifying the end of the transfer. This short packet is not transferred by the DMA controller: instead the USB controller interrupts the processor by generating the appropriate endpoint interrupt. The processor can then read the USB_RXCOUNT register to see the size of the short packet and either unload it manually or reprogram the DMA controller in mode 0 to unload the packet.

The DMAxADDR register is incremented as the packets are unloaded, so the processor can determine the size of the transfer by comparing the current value of DMAxADDR with the start address of the memory buffer.

If the size of the transfer exceeds the data buffer size, the DMA controller stops unloading the FIFO and interrupts the processor.

Multiple Packets: TX Endpoints

Using DMA mode 1 for a TX endpoint, the DMA controller is programmed as follows:

- DMAxADDR: memory address of data block to send
- USB_DMAxCOUNT: size of data block
- USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 1, DMAREQMODE_T = 1

The USB controller TX endpoint is programmed as follows:

1. The relevant EPx_TX_E bit in the USB_INTRTXE register is set to 1.
2. The AUTOSET_T and DMA_ENA bits of the appropriate USB_EP_NIx_TXCSR register is set to 1.

When the FIFO in the USB controller becomes available, the DMA controller requests bus mastership and transfers a packet to the FIFO. With AUTOSET_T set, the USB controller automatically sets the TXPKTRDY bit. This process continues until the entire data block is transferred to the USB controller. The DMA controller then interrupts the processor by taking DMAx_INT low. If the last packet to be loaded was less than the maximum packet size for the endpoint, the TXPKTRDY bit is not set for this packet; the processor should respond to the DMA interrupt by setting the TXPKTRDY bit to allow the last short packet to be sent. If the last packet to be loaded was of the maximum packet size, then the action to take depends on whether the transfer is under the control of an application such as the mass storage software on Windows system that keeps count of the individual packets sent. If the transfer is not under such control, the processor should still respond to the DMA interrupt by setting the TXPKTRDY bit. This has the effect of sending a null packet for the receiving software to interpret as indicating the end of the transfer.

USB OTG Registers

The USB OTG has a number of memory-mapped registers (MMRs) that regulate its operation. Descriptions and bit diagrams for most of these registers are provided in the following sections. See Table A-22 on page A-42 for a complete list of USB-OTG registers and their addresses.

USB Global Control (USB_GLOBAL_CTL) Register

The USB_GLOBAL_CTL register (see Figure 26-34) enables software control of the internal clocking of the USB. This control permits reducing power consumption by minimizing switching activity in endpoint logic, which is not required for use.

Before an endpoint can be used for transfer on USB it must first be activated by setting the appropriate bit in the USB_GLOBAL_CTL register. The GLOBAL_ENA bit must be set any time the USB controller is required for use. The GLOBAL_ENA bit also brings the USB PHY and USB PLL out of reset state. The USB PLL locks with the frequency multiplier value programmed in the USB_PLLOSC_CTRL register. When USB_GLOBAL_CTL is not configured, the behavior of the USB controller is undefined and writes into CSR registers and FIFOs are not committed. It is not possible to access an endpoint FIFO location when that endpoint is not activated in this register. Similarly, the GLOBAL_ENA bit is required for access to the endpoint 0 FIFO locations. For more information on the USB controller clocking scheme, see "Power and Clocking" on page 26-55.

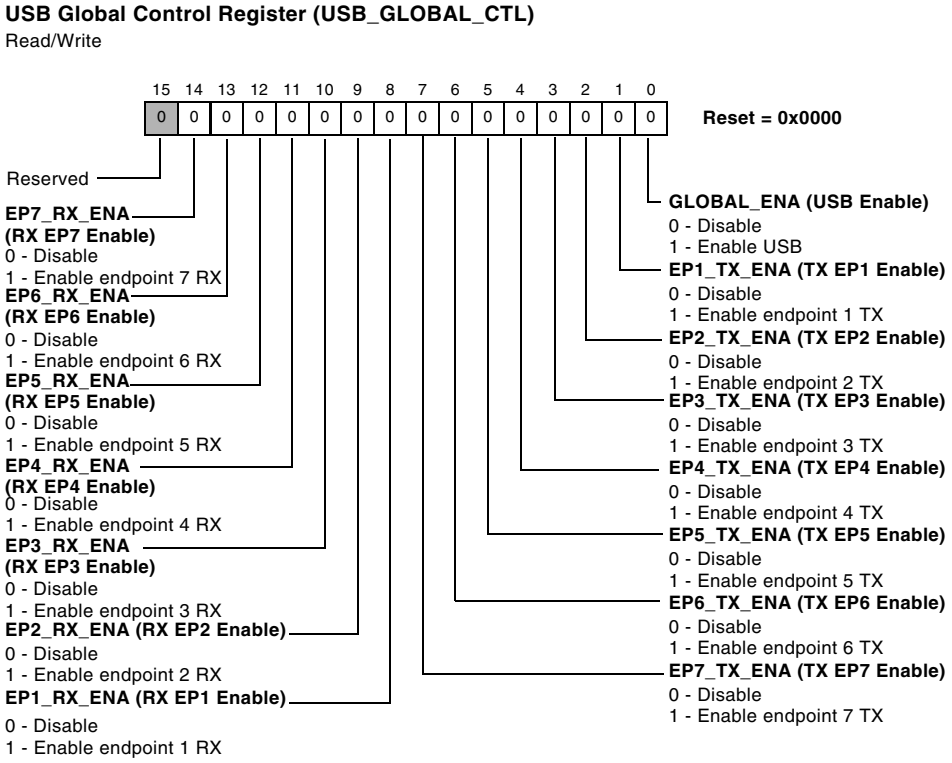


Figure 26-34. USB Global Control Register

Bit 15, which is marked as reserved in Figure 26-34, implements the test mode timer reduction. When set, this bit reduces the values used in the timers internal to the USB protocol block in order to drastically reduce the simulation time. This bit should only be set for simulation purposes, because setting it causes incorrect USB behavior if set during normal operation.

USB Power Management (USB_POWER) Register

The USB_POWER register (see Figure 26-35) controls suspend and resume signaling and controls some operational aspects of the USB controller.

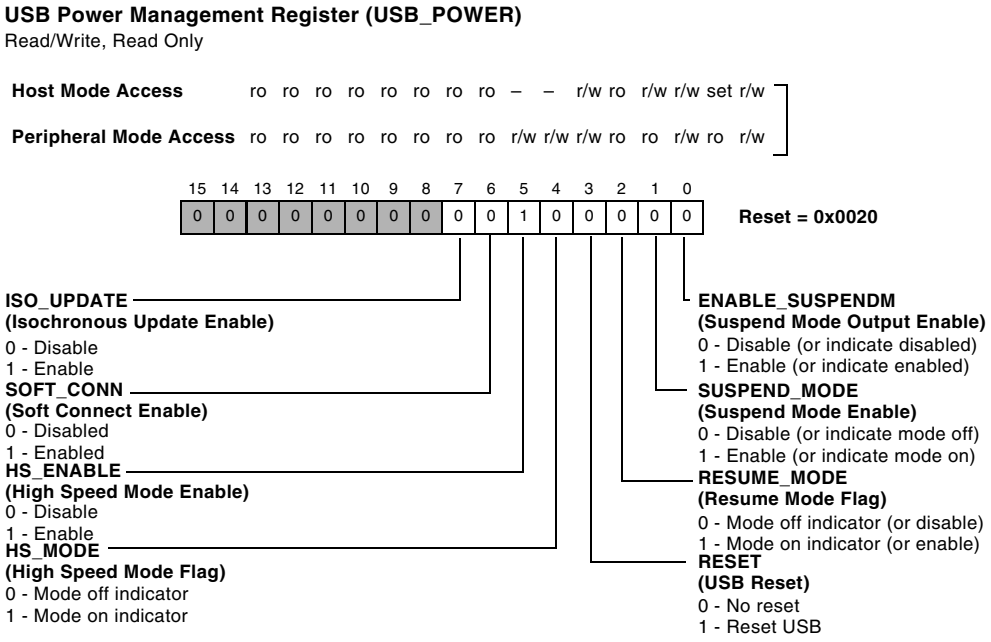


Figure 26-35. USB Power Management Register

ENABLE_SUSPENDM

The ENABLE_SUSPENDM (bit 0) is set by the processor core to enable the SUSPENDM output (internal USB controller signal). When this bit is set, the SUSPENDM output signal is used by the USB PHY to power-down its drivers when the USB controller is not active.

SUSPEND_MODE

In host mode, SUSPEND_MODE (bit 1) is set by the processor core to enter suspend mode. In peripheral mode, this bit is set on entry into suspend mode. It is cleared when the processor core reads the interrupt register, or sets the resume bit.

RESUME_MODE

The RESUME_MODE (bit 2) is set by the processor core to generate resume signaling when the function is in suspend mode. The processor core should clear this bit after 10 ms (a maximum of 15 ms) to end resume signaling. In host mode, this bit is also automatically set when resume signaling from the target is detected while the USB controller is suspended.

RESET

The RESET (bit 3) bit is set when reset signaling is present on the bus. This bit is read/write from the processor core in host mode but read-only in peripheral mode.

HS_MODE

When HS_MODE (bit 4) is set, this read-only bit indicates high-speed mode successfully negotiated during a USB reset. In peripheral mode, it becomes valid when the USB reset completes (as indicated by the USB reset interrupt). In host mode, it becomes valid when the RESET_OR_BABLE_B bit is cleared. It remains valid for the duration of the session.

HS_ENABLE

When HS_ENABLE (bit 5) is set by the processor core, the USB controller negotiates for high speed when the device is reset by the hub/host. If it is not set, the controller only operates in full-speed mode. By default HS_ENABLE is set to 1.

SOFT_CONN

If the soft connect/disconnect feature is enabled (bit 6, SOFT_CONN = 1), then the USB D+/D–lines are enabled when this bit is set by the processor core and three-stated when this bit is cleared by the processor core. Only valid in peripheral mode.

ISO_UPDATE

When ISO_UPDATE (bit 7) is set by the processor core, the USB controller waits for an SOF token from the time TXPKTRDY is set before sending the packet. If an IN token is received before an SOF token, then a zero length data packet is sent. Only valid in peripheral mode. Also, this bit only affects endpoints performing isochronous transfers.

USB Function Address (USB_FADDR) Register

The USB_FADDR register (see Figure 26-36) contains the 7-bit address of the peripheral part of the transaction.

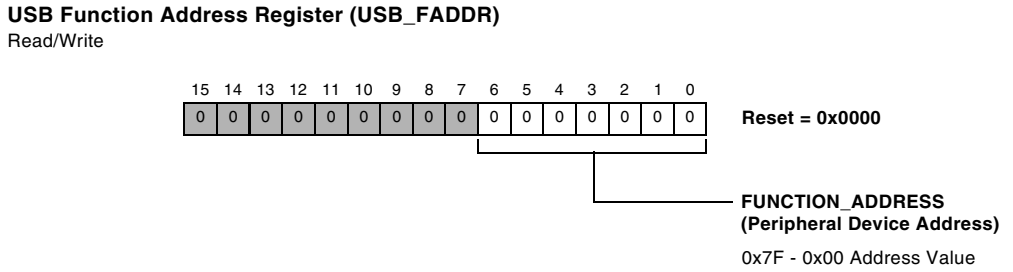


Figure 26-36. USB Function Address Register

When the USB controller is being used in peripheral mode (HOST_MODE = 0 in USB_OTG_DEV_CTL), this register is written with the address received through a SET_ADDRESS command. The address is used for decoding the function address in subsequent token packets.

When the USB controller is being used in host mode (HOST_MODE = 1 in USB_OTG_DEV_CTL), this register is set to the value sent in a SET_ADDRESS command during device enumeration as the address for the peripheral device.

USB Test Mode (USB_TESTMODE) Register

The USB_TESTMODE register (see Figure 26-37) places the USB controller into test mode state and also can put the USB controller into one of the four test modes for high-speed operation (see the USB 2.0 specification).

USB Test Mode Register (USB_TESTMODE)
Read/Write

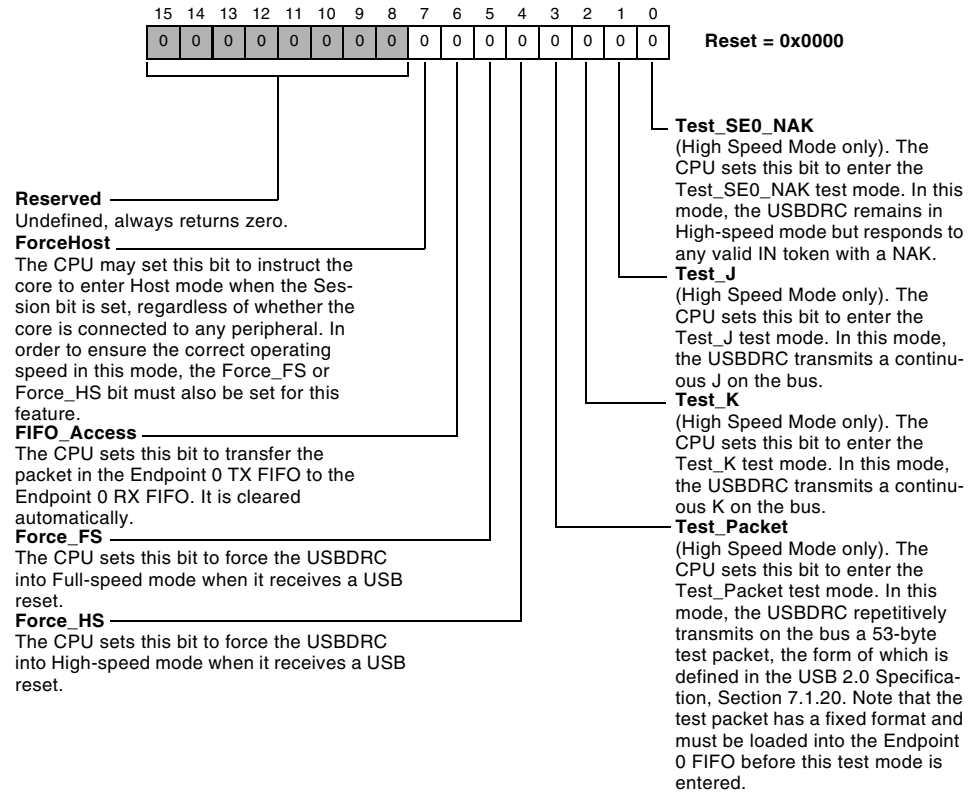


Figure 26-37. USB Test Mode Register

USB_TESTMODE is not used in normal operation. Only one of the bits may be set at one time, except for bit 5 in conjunction with the ForceHost feature.

USB Global Interrupt (USB_GLOBINTR) Register

The USB_GLOBINTR register (see Figure 26-38) selects routing for each of the three USB interrupt sources (USB_INTRRX, USB_INTRTX and USB_INTRUSB/USB_OTG_VBUS_IRQ) to any or all of the top-level interrupts (USB_INT0, USB_INT1 and USB_INT2).

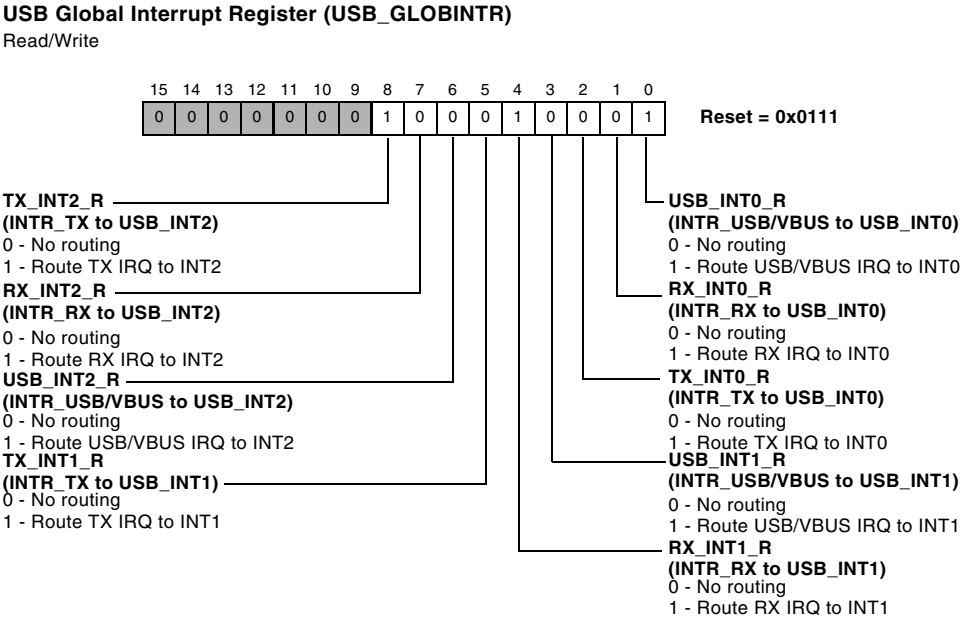


Figure 26-38. USB Global Interrupt Register

Each interrupt source is represented by a configuration bit across each of the top-level interrupts. Setting each to a 1, routes that source to the interrupt.

USB Transmit Interrupt (USB_INTRTX) Register

The USB_INTRTX register (see Figure 26-39) indicates which interrupts are currently active for endpoint 0 and the TX endpoints 1–7. Writing 1 to bits 0–7 when they are high clears that particular bit and de-asserts the corresponding interrupt source.

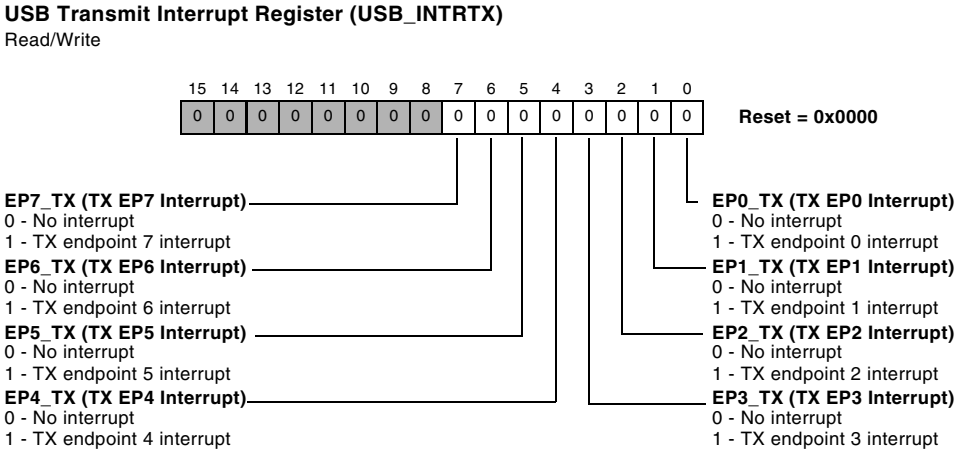


Figure 26-39. USB Transmit Interrupt Register

USB Receive Interrupt (USB_INTRRX) Register

The USB_INTRRX register (see Figure 26-40) indicates which interrupts are currently active for the RX endpoints 1–7. Writing 1 to bits 1–7 when they are high clears that particular bit and de-asserts the corresponding interrupt source.

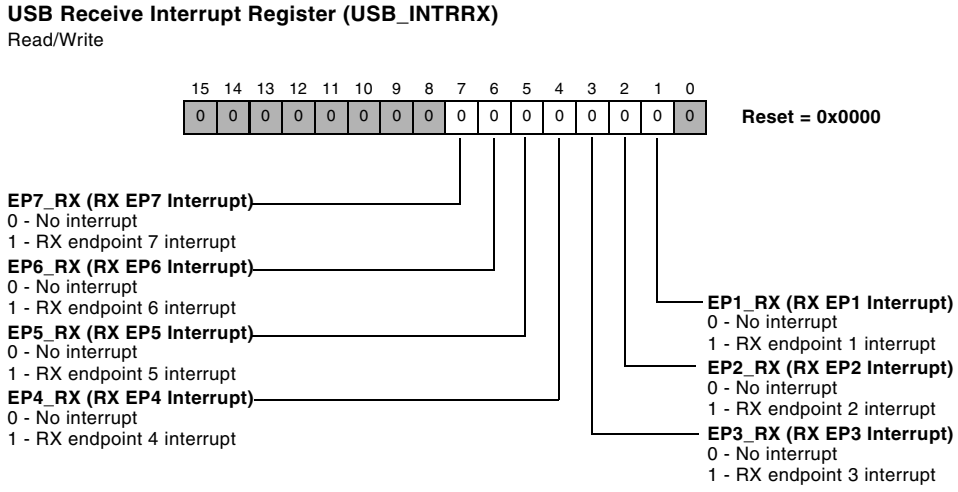


Figure 26-40. USB Receive Interrupt Register

USB Transmit Interrupt Enable (USB_INTRTXE) Register

The USB_INTRTXE register (see Figure 26-41) enables interrupts for endpoint 0 and the TX endpoints 1–7.

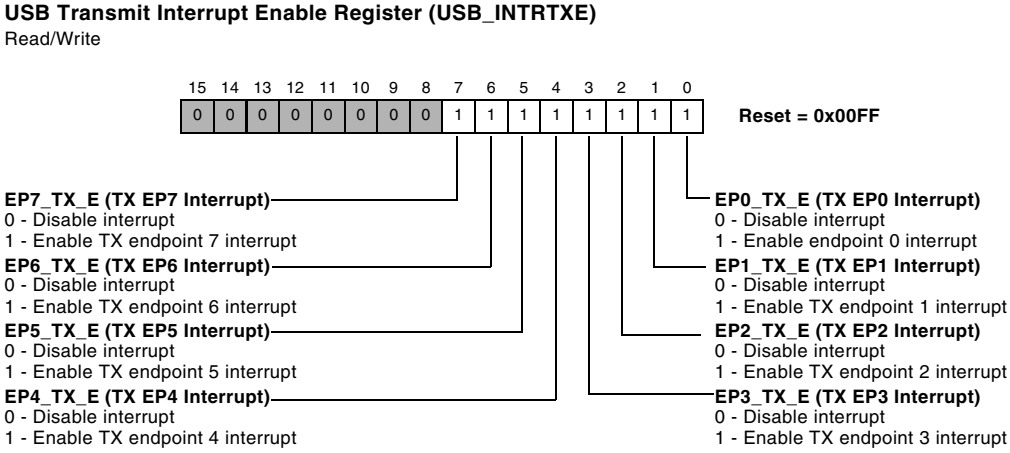


Figure 26-41. USB Transmit Interrupt Enable Register

Writing 1 to bits 0–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 0–7 disables (masks) an interrupt source. The corresponding status bit in the USB_INTRTX register may still be set, but no interrupt is asserted. On reset, the bits corresponding to endpoint 0 and the TX endpoints included in the design are set to 1 (for example, all TX interrupts are enabled).

USB Receive Interrupt Enable (USB_INTRRXE) Register

The USB_INTRRXE register (see Figure 26-42) enables interrupts for the RX endpoints 1–7.

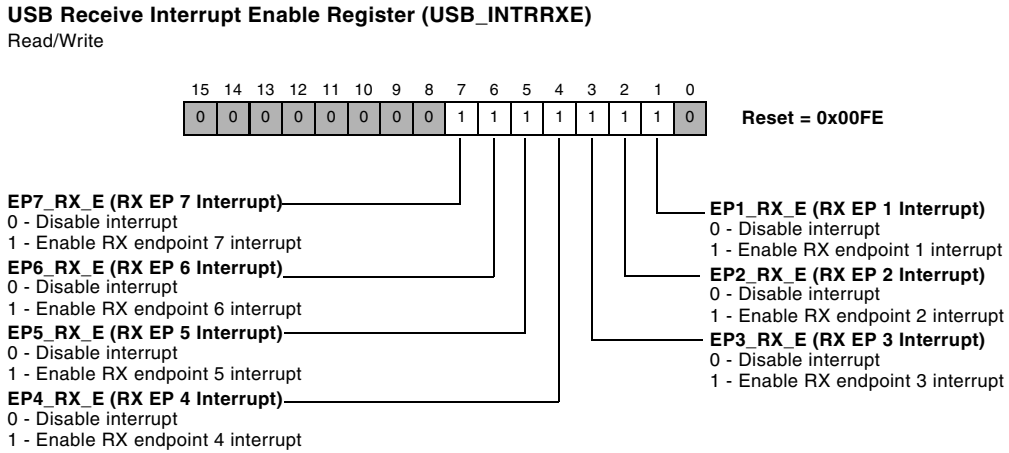


Figure 26-42. USB Receive Interrupt Enable Register

Writing 1 to bits 1–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 1–7 disables (masks) an interrupt source. The corresponding status bit in the USB_INTRRX register may still be set, but no interrupt is asserted. On reset, the bits corresponding to endpoint 0 and the TX endpoints included in the design are set to 1 (for example, all TX interrupts are enabled).

USB Common Interrupts (USB_INTRUSB) Register

The USB_INTRUSB register (see Figure 26-43) indicates which USB interrupts are currently active.

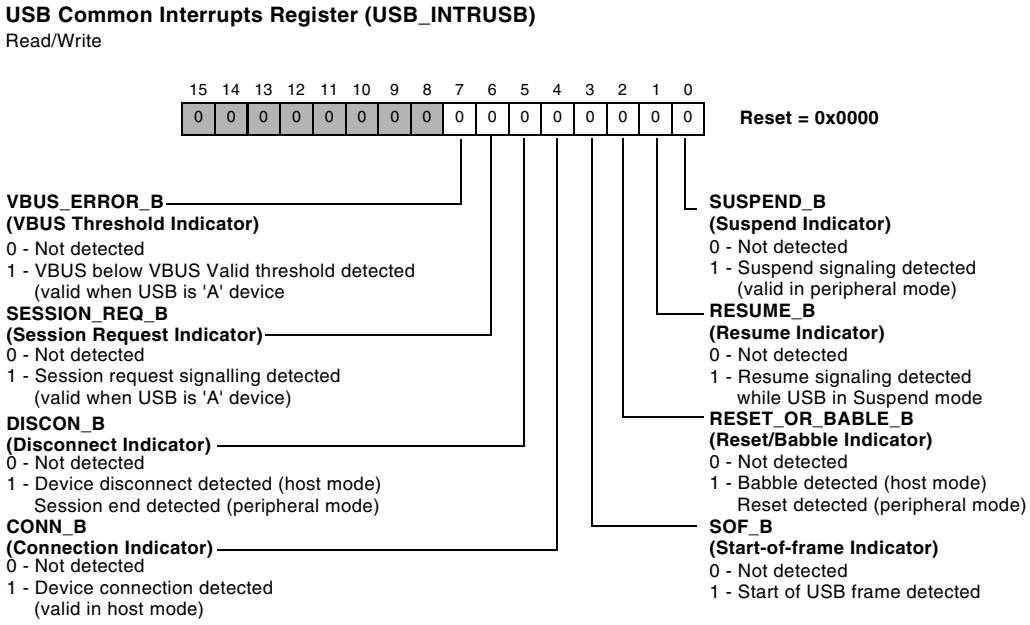


Figure 26-43. USB Common Interrupts Register

Writing a 1 to any of the bits 0–7 when they are high de-asserts the interrupt source corresponding to that bit. The USB_INTRUSB register shares an interrupt source line with USB_OTG_VBUS_IRQ.

USB Common Interrupt Enable (USB_INTRUSBE) Register

The USB_INTRUSBE register (see Figure 26-44) enables common USB interrupts. Writing 1 to bits 0–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 0–7 disables (masks) an interrupt source. The corresponding status bit in the USB_INTUSB register may still be set, but no interrupt is asserted. On reset, the RESUME_BE and RESET_OR_BABLE_BE bits are set to 1 (for example, interrupts for resume signalling detection and reset/babble detection are enabled).

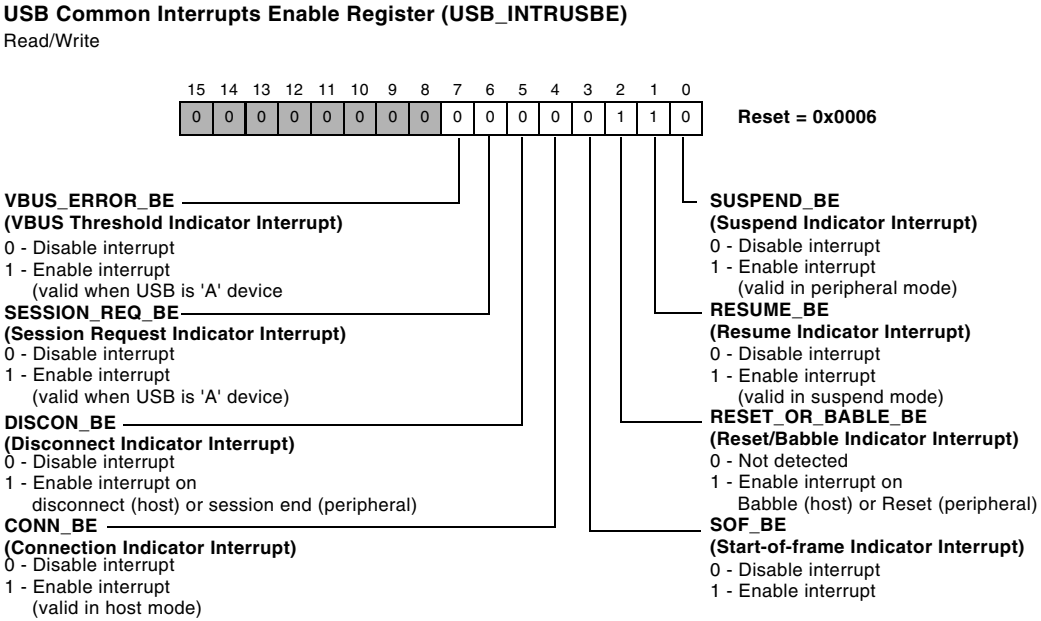


Figure 26-44. USB Common Interrupts Enable Register

USB Frame Number (USB_FRAME) Register

The USB_FRAME register (see Figure 26-45) contains the last received frame number; bit 10 is MSB; bit 0 is LSB.

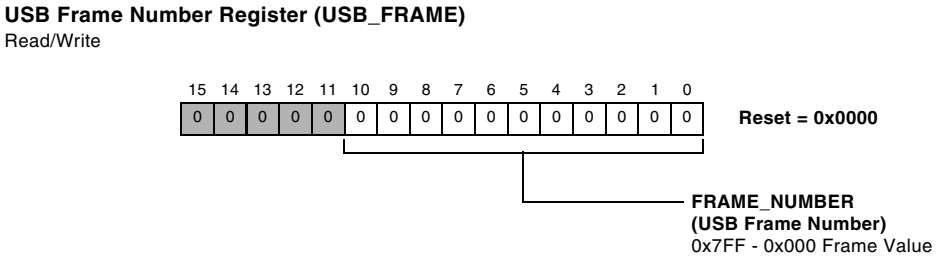


Figure 26-45. USB Frame Number Register

USB Index (USB_INDEX) Register

The USB_INDEX register (see Figure 26-46) contains an index value for alternate addressing of USB endpoint control and status registers.

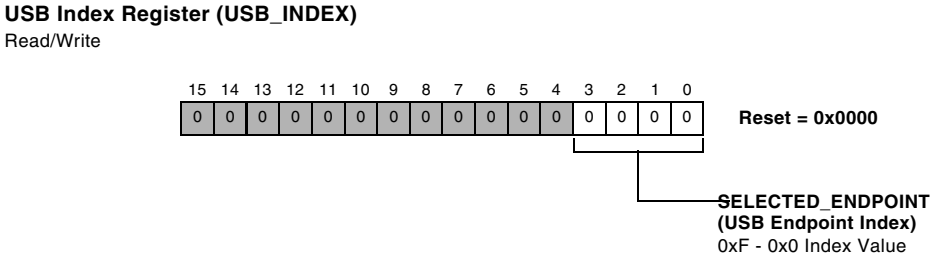


Figure 26-46. USB Index Register

Each TX endpoint and each RX endpoint have their own set of control/status registers located between address 0xFFC0 3E00 and 0xFFC0 3FF8. In addition, one indexed set of TX control/status and one set of RX control/status registers appear between address 0xFFC0 3C40

and 0xFFC0 3C68. The USB_INDEX is a 4-bit register that determines which set of endpoint control/status registers are accessed at the indexed address range.

Before accessing an endpoint's control/status registers using the indexed range, the endpoint number is written to the USB_INDEX register to ensure that the correct control/status registers appear in the indexed range of the memory map.

USB TX Max Packet (USB_TX_MAX_PACKET) Register

The USB_TX_MAX_PACKET register (see Figure 26-47) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame. When setting this value, you must consider the constraints placed by the USB specification on packet sizes for bulk, interrupt and isochronous transactions in full-speed operations. The USB_TX_MAX_PACKET register provides indexed access to the USB_EP_NIx_TXMAXP register for each TX endpoint (except endpoint 0).

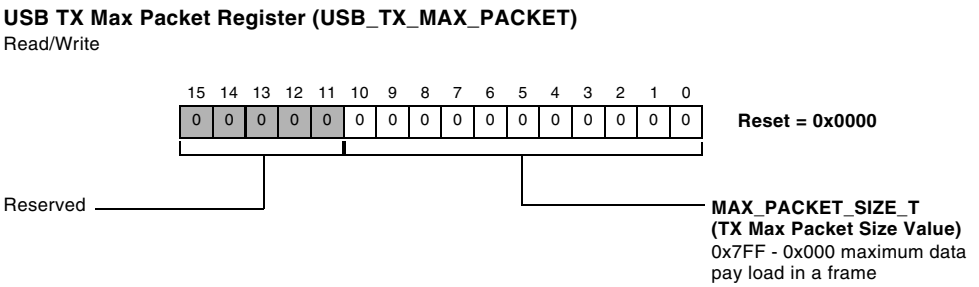


Figure 26-47. USB TX Max Packet Register

USB Control/Status EP0 (USB_CSR0) Register

The USB_CSR0 register (see Figure 26-48) provides control and status bits for endpoint 0. Note that some bits may be set to clear automatically. The interpretation of the USB_CSR0 register depends on whether the USB controller is acting as a peripheral or as a host.

Many bits in this register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

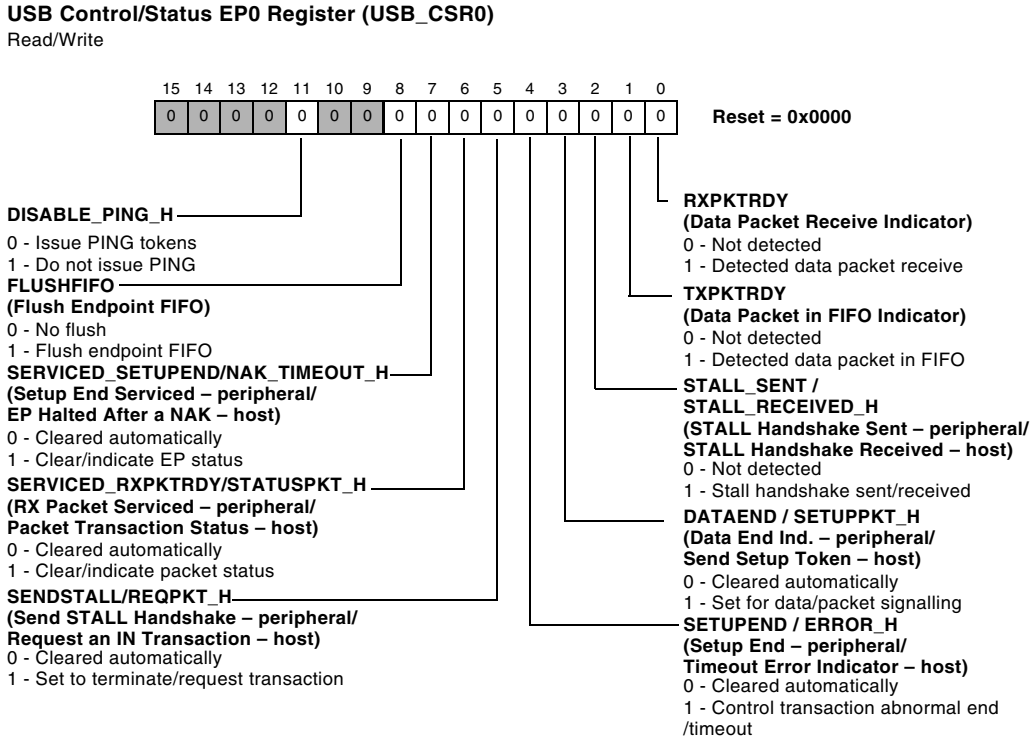


Figure 26-48. USB Control/Status EP0 Register

RXPKTRDY

In peripheral mode, RXPKTRDY (bit 0) is set when a data packet is received. An interrupt is generated when this bit is set. The processor core clears this bit by setting the SERVICED_RXPKTRDY bit.

In host mode, RXPKTRDY (bit 0) is set when a data packet is received. An interrupt is generated (if enabled) when this bit is set. The processor core should clear this bit when the packet is read from the FIFO.

TXPKTRDY

In peripheral mode, the processor core sets TXPKTRDY (bit 1) after loading a data packet into the FIFO. It is cleared automatically when the data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

In host mode, the processor core sets TXPKTRDY (bit 1) after loading a data packet into the FIFO. It is cleared automatically when the data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

STALL_SENT / STALL_RECEIVED_H

In peripheral mode, STALL_SENT (bit 2) is set when a STALL handshake is transmitted. The processor core should clear this bit.

In host mode, STALL_RECEIVED_H (bit 2) is set when a STALL handshake is received. The processor core should clear this bit.

DATAEND / SETUPPKT_H

In peripheral mode, the processor core sets DATAEND (bit 3):

1. When setting TXPKTRDY for the last data packet.
2. When clearing RXPKTRDY after unloading the last data packet.
3. When setting TXPKTRDY for a zero length data packet. It is cleared automatically.

In host mode, the processor core sets SETUPPKT_H (bit 3), at the same time as the TXPKTRDY bit is set, to send a SETUP token instead of an OUT token for the transaction.

SETUPEND / ERROR_H

In peripheral mode, SETUPEND (bit 4) is set when a control transaction ends before the DATAEND bit is set. An interrupt is generated and the FIFO is flushed at this time. The bit is cleared by the processor core writing a 1 to the SERVICED_SETUPEND bit.

In host mode, ERROR_H (bit 4) is set when three attempts have been made to perform a transaction with no response from the peripheral. The processor core should clear this bit. An interrupt is generated when this bit is set.

SENDSTALL / REQPKT_H

In peripheral mode, the processor core writes a 1 to SENDSTALL (bit 5) to terminate the current transaction. The STALL handshake is transmitted, then this bit automatically is cleared.

In host mode, the processor core sets REQPKT_H (bit 5) to request an IN transaction. It is cleared when RXPKTRDY is set.

SERVICED_RXPKTRDY / STATUSPKT_H

In peripheral mode, the processor core writes a 1 to SERVICED_RXPKTRDY (bit 6) to clear the RXPKTRDY bit. It is cleared automatically.

In host mode, the processor core sets STATUSPKT_H (bit 6) at the same time as the TXPKTRDY or REQPKT_H bit is set, to perform a status stage transaction. Setting this bit ensures that the data toggle is set to 1 so that a DATA1 packet is used for the Status Stage transaction.

SERVICED_SETUPEND / NAK_TIMEOUT_H

In peripheral mode, the processor core writes a 1 to SERVICED_SETUPEND (bit 7) to clear the SETUPEND bit. It is cleared automatically.

In host mode, NAK_TIMEOUT_H (bit 7) is set when endpoint 0 is halted following the receipt of NAK responses for longer than the time set by the USB_NAKLIMIT0 register. The processor core should clear this bit to allow the endpoint to continue.

FLUSHFIFO

In peripheral mode, the processor core writes a 1 to the FLUSHFIFO (bit 8) to flush the next packet to be transmitted/read from the endpoint 0 FIFO. The FIFO pointer is reset and the TXPKTRDY or RXPKTRDY bit (below) is cleared. FLUSHFIFO has no effect unless TXPKTRDY or RXPKTRDY is set.

In host mode, the processor core writes a 1 to FLUSHFIFO (bit 8) to flush the next packet to be transmitted/read from the endpoint 0 FIFO. The FIFO pointer is reset and the TXPKTRDY or RXPKTRDY bit (below) is cleared. FLUSHFIFO has no effect unless TXPKTRDY or RXPKTRDY is set.

DISABLE_PING_H

The processor core writes a 1 to this bit to instruct the USB controller not to issue PING tokens in data and status phases of a high-speed control transfer (for use with devices that do not respond to PING).

USB TX Control/Status EPx (USB_TXCSR) Register

The USB_TXCSR register (see Figure 26-49) provides control and status bits for transfers through the currently selected TX endpoint.

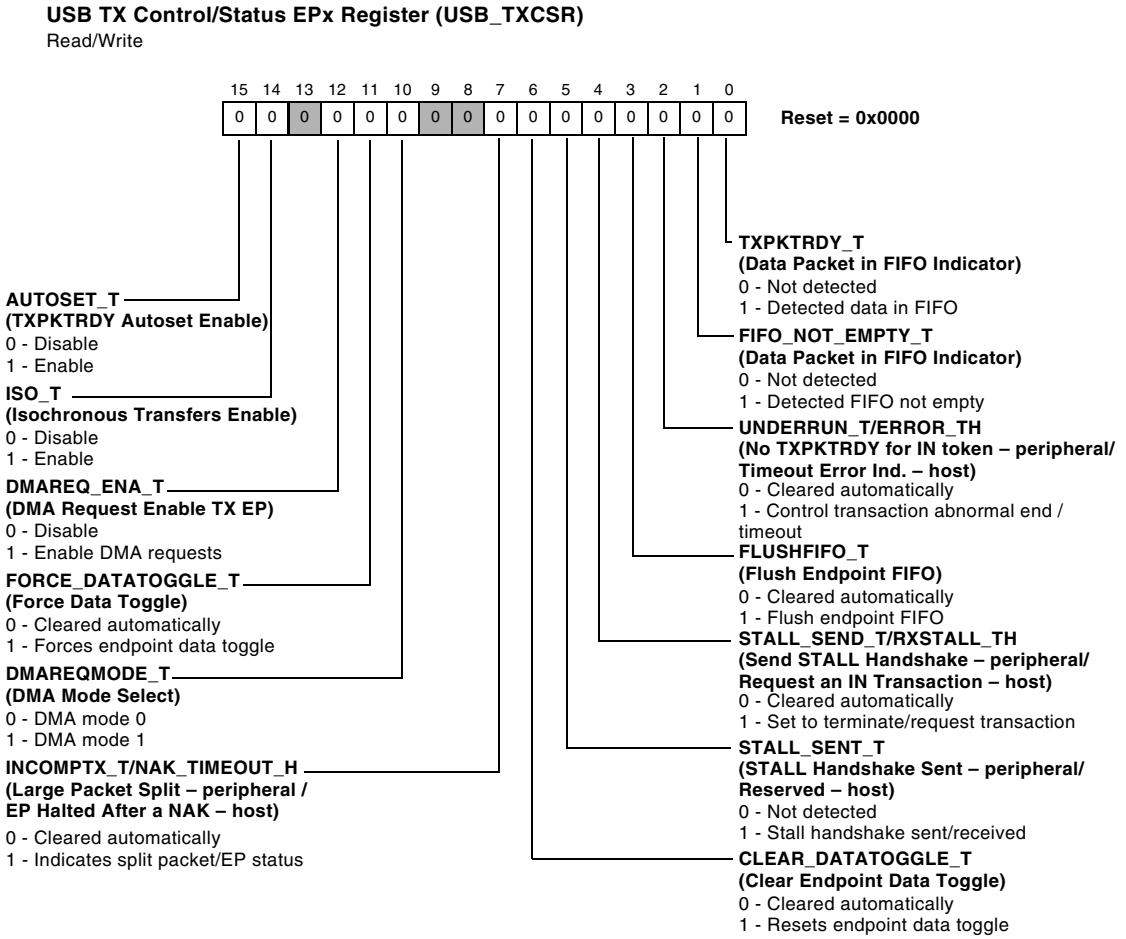


Figure 26-49. USB TX Control/Status EPx Register

Note that some bits may be set to clear automatically. The interpretation of the USB_TXCSR register depends on whether the USB controller is acting as a peripheral or as a host.

There is a USB_EP_NIx_TXCSR register for each TX endpoint, except endpoint 0. These registers may be accessed directly through the register address or through the USB_TXCSR register indexed by the USB_INDEX register.

Many bits in the USB_TXCSR register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

TXPKTRDY_T

In peripheral mode, the processor core sets TXPKTRDY_T (bit 0) after loading a data packet into the FIFO. It is cleared automatically when a data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

In host mode, the processor core sets TXPKTRDY_T (bit 0) after loading a data packet into the FIFO. It is cleared automatically when a data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

FIFO_NOT_EMPTY_T

In peripheral mode, the USB sets FIFO_NOT_EMPTY_T (bit 1) when there is at least one packet in the TX FIFO.

In host mode, the USB sets FIFO_NOT_EMPTY_T (bit 1) when there is at least one packet in the TX FIFO.

UNDERRUN_T / ERROR_TH

In peripheral mode, the USB sets UNDERRUN_T (bit 2) if an IN token is received when TXPKTRDY is not set. The processor core should clear this bit.

In host mode, the USB sets ERROR_TH (bit 2) when three attempts have been made to send a packet and no handshake packet is received. The processor core should clear this bit. An interrupt is generated when the bit is set. Valid only when the endpoint is operating in bulk or interrupt mode.

FLUSHFIFO_T

In peripheral mode, the processor core writes a 1 to FLUSHFIFO_T (bit 3) to flush the next packet to be transmitted from the endpoint TX FIFO. The FIFO pointer is reset and the TXPKTRDY bit (below) is cleared. This pointer may be set simultaneously with TxPktRdy to abort the packet that is currently being loaded into the FIFO. FLUSHFIFO_T has no effect unless TXPKTRDY is set. Note that if the FIFO is double-buffered, FLUSHFIFO_T may need to be set twice to completely clear the FIFO.

In host mode, the processor core writes a 1 to FLUSHFIFO_T (bit 3) to flush the next packet to be transmitted from the endpoint TX FIFO. The FIFO pointer is reset and the TXPKTRDY bit (below) is cleared. This pointer may be set simultaneously with TxPktRdy to abort the packet that is currently being loaded into the FIFO. FLUSHFIFO_T has no effect unless TXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_T may need to be set twice to completely clear the FIFO.

STALL_SEND_T / STALL_RECEIVED_TH

In peripheral mode, the processor core writes a 1 to STALL_SEND_T (bit 4) to issue a STALL handshake to an IN token. The processor core clears this bit to terminate the stall condition. This bit has no effect where the endpoint is being used for isochronous transfers.

In host mode, bit 4 is reserved.

STALL_SENT_T / RXSTALL_TH

In peripheral mode, SENTSTALL (bit 5) is set when a STALL handshake is transmitted. The FIFO is flushed and the TXPKTRDY bit is cleared. The processor core should clear this bit.

In host mode, RXSTALL_TH (bit 5) is set when a STALL handshake is received. The FIFO is flushed and the TXPKTRDY bit is cleared. The processor core should clear this bit.

CLEAR_DATATOGGLE_T

In peripheral mode, the processor core writes a 1 to CLEAR_DATATOGGLE_T (bit 6) to reset the endpoint data toggle to 0.

In host mode, the processor core writes a 1 to CLEAR_DATATOGGLE_T (bit 6) to reset the endpoint data toggle to 0.

INCOMPTX_T / NAK_TIMEOUT_TH

In peripheral mode, this bit always returns 0.

In host mode, NAK_TIMEOUT_TH (bit 7) is set when the TX endpoint is halted following the receipt of NAK responses for longer than the time set as the NAK limit by the USB_TXINTERVAL register. The processor core should clear this bit to allow the endpoint to continue. This bit is valid only for bulk endpoints.

DMAREQMODE_T

In peripheral mode, the processor core sets DMAREQMODE_T (bit 10) to select DMA mode 1 and clears this bit to select DMA mode 0.

In host mode, the processor core sets DMAREQMODE_T (bit 10) to select DMA mode 1 and clears this bit to select DMA mode 0.

FORCE_DATATOGGLE_T

In peripheral mode, the processor core sets FORCE_DATATOGGLE_T (bit 11) to force the endpoint data toggle to switch and the data packet to be cleared from the FIFO, regardless of whether an ACK was received. This can be used by interrupt TX endpoints that are used to communicate rate feedback for isochronous endpoints.

In host mode, the processor core sets FRCDATATOG (bit 11) to force the endpoint data toggle to switch and the data packet to be cleared from the FIFO, regardless of whether an ACK was received. This can be used by interrupt TX endpoints that are used to communicate rate feedback for isochronous endpoints.

DMAREQ_ENA_T

In peripheral mode, the processor core sets DMAREQ_ENA_T (bit 12) to enable the DMA request for the TX endpoint.

In host mode, the processor core sets DMAREQ_ENA_T (bit 12) to enable the DMA request for the TX endpoint.

ISO_T

In peripheral mode, the processor core sets ISO_T (bit 14) to enable the TX endpoint for isochronous transfers, and clears it to enable the TX endpoint for bulk or interrupt transfers. This bit only has an effect in peripheral mode.

In host mode, bit 14 is unused, and always returns zero.

AUTOSET_T

In peripheral mode, if the processor core sets AUTOSET_T (bit 15), TXPKTRDY automatically is set when data of the maximum packet size (value in USB_TX_MAX_PACKET) is loaded into the TX FIFO. If a packet of less than the maximum packet size is loaded, then TXPKTRDY must be set manually.

In host mode, if the processor core sets AUTOSET_T (bit 15), TXPKTRDY automatically is set when data of the maximum packet size (value in USB_TX_MAX_PACKET) is loaded into the TX FIFO. If a packet of less than the maximum packet size is loaded, then TXPKTRDY must be set manually.

USB RX Max Packet (USB_RX_MAX_PACKET) Register

The USB_RX_MAX_PACKET register (see Figure 26-50) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame.

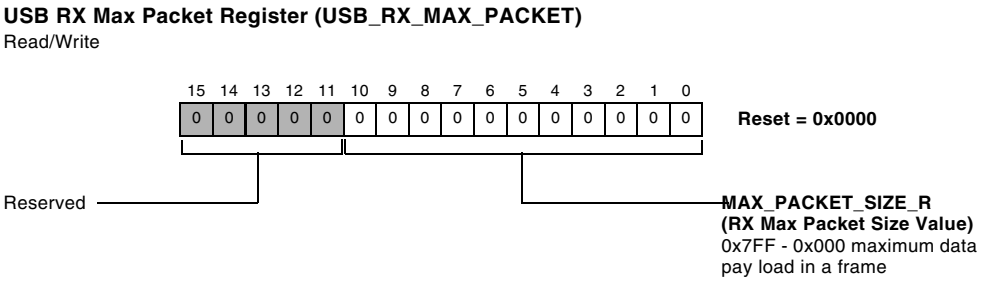


Figure 26-50. USB RX Max Packet Register

The USB_RX_MAX_PACKET register provides indexed access to the USB_EP_NIx_RXMAXP register for each RX endpoint (except endpoint 0). Bits[10:0] define (in bytes) the maximum payload transmitted in a single transaction. The legal value loaded can be up to 1023 bytes but is subject to the constraints placed by the USB specification on packet sizes for bulk, interrupt and isochronous transfers in full-speed operation.

A value greater than the maximum allowed of 1023 for full-speed USB operation produces unpredictable results.

The value written to this register should match the programmed maximum individual packet size (*MaxPktSize*) of the standard endpoint descriptor for the associated endpoint (see *Universal Serial Bus Specification Revision 2.0*, Chapter 9). A mismatch could cause unexpected results.

The total amount of data represented by the value written to this register must not exceed the RX FIFO size, and should not exceed half the FIFO size if double-buffering is required.

USB RX Control/Status (USB_RXCSR) Register

The USB_RXCSR register (see Figure 26-51) provides control and status bits for transfers through the currently selected RX endpoint.

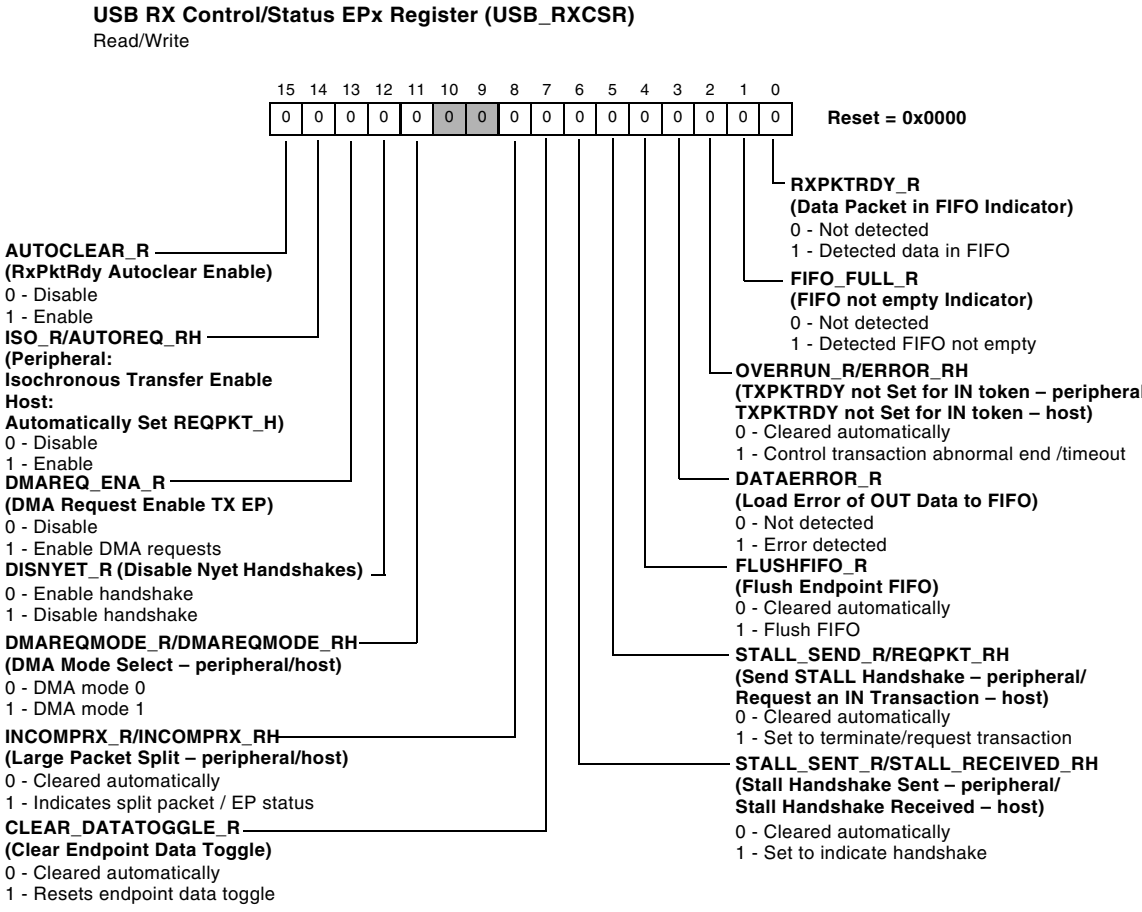


Figure 26-51. USB RX Control/Status EPx Register

Note that some bits may be set to clear automatically. The interpretation of the USB_RXCSR register depends on whether the USB controller is acting as a peripheral or as a host.

There is a USB_EP_NIx_RXCSR register for each RX endpoint, except endpoint 0. These registers may be accessed directly through the register address or through the USB_RXCSR register indexed by the USB_INDEX register.

Many bits in the USB_RXCSR register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

RXPKTRDY_R

In peripheral mode, RXPKTRDY_R (bit 0) is set when a data packet is received. The processor core should clear this bit when the packet is unloaded from the RX FIFO. An interrupt is generated when the bit is set.

In host mode, RXPKTRDY_R (bit 0) is set when a data packet is received. The processor core should clear this bit when the packet is unloaded from the RX FIFO. An interrupt is generated when the bit is set.

FIFO_FULL_R

In peripheral mode, FIFO_FULL_R (bit 1) is set when no more packets can be loaded into the RX FIFO.

In host mode, FIFO_FULL_R (bit 1) is set when no more packets can be loaded into the RX FIFO.

OVERRUN_R / ERROR_RH

In peripheral mode, OVERRUN_R (bit 2) is set if an OUT packet cannot be loaded into the RX FIFO. The processor core should clear this bit. This bit is only valid when the endpoint is operating in isochronous mode. In bulk mode, it always returns zero.

In host mode, the USB sets ERROR_RH (bit 2) when 3 attempts have been made to receive a packet and no data packet is received. The processor core should clear this bit. An interrupt is generated when the bit is set. This bit is only valid when the TX endpoint is operating in bulk or interrupt mode. In isochronous mode, it always returns zero.

DATAERROR_R

In peripheral mode, DATAERROR_R (bit 3) is set when RXPKTRDY is set if the data packet has a CRC or bit-stuff error. It is cleared when RXPKTRDY is cleared. This bit is only valid when the endpoint is operating in isochronous mode. In bulk mode, it always returns zero.

In host mode, when operating in isochronous mode, DATAERROR_R (bit 3) is set when RXPKTRDY is set and the data packet has a CRC or bit-stuff error and cleared when RXPKTRDY is cleared. In bulk mode, this bit is set when the RX endpoint is halted following the receipt of NAK responses for longer than the time set as the NAK limit by the USB_RXINTERVAL register. The processor core should clear this bit to allow the endpoint to continue.

FLUSHFIFO_R

In peripheral mode, the processor core writes a 1 to FLUSHFIFO_R (bit 4) to flush the next packet to be read from the endpoint RX FIFO. The FIFO pointer is reset and the RXPKTRDY bit (below) is cleared. FLUSHFIFO_R has no effect unless RXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_R may need to be set twice to completely clear the FIFO.

In host mode, the processor core writes a 1 to FLUSHFIFO_R (bit 4) to flush the next packet to be read from the endpoint RX FIFO. The FIFO pointer is reset and the RXPKTRDY bit (below) is cleared. FLUSHFIFO_R has no effect unless RXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_R may need to be set twice to completely clear the FIFO.

STALL_SEND_R / REQPKT_RH

In peripheral mode, the processor core writes a 1 to STALL_SEND_R (bit 5) to issue a STALL handshake. The processor core clears this bit to terminate the stall condition. This bit has no effect where the endpoint is being used for isochronous transfers.

In host mode, the processor core writes a 1 to REQPKT_RH (bit 5) to request an IN transaction. It is cleared when RXPKTRDY is set.

STALL_SENT_R / STALL_RECEIVED_RH

In peripheral mode, STALL_SENT_R (bit 6) is set when a STALL handshake is transmitted. The processor core should clear this bit.

In host mode, when a STALL handshake is received, STALL_RECEIVED_RH (bit 6) is set and an interrupt is generated. The processor core should clear this bit.

CLEAR_DATATOGGLE_R

In peripheral mode, the processor core writes a 1 to CLEAR_DATATOGGLE_R (bit 7) to reset the endpoint data toggle to 0.

In host mode, the processor core writes a 1 to CLEAR_DATATOGGLE_R (bit 7) to reset the endpoint data toggle to 0.

INCOMPRX_R / INCOMPRX_RH

In peripheral mode, INCOMPRX_R (bit 8) always returns 0.

In host mode, INCOMPRX_RH (bit 8) always returns 0.

DMAREQMODE_R / DMAREQMODE_RH

In peripheral mode, the processor core sets DMAREQMODE_R (bit 11) to select DMA request mode 1 and clears this bit to select DMA request mode 0.

In host mode, the processor core sets DMAREQMODE_RH (bit 11) to select DMA mode 1 and clears this bit to select DMA mode 0.

DISNYET_R

In peripheral mode, the processor core sets DISNYET_R (bit 12) to disable the sending of NYET handshakes. When set, all successfully received RX packets are acknowledged, including at the point at which the FIFO becomes full. This bit only has an effect in high-speed mode, where it is set for all interrupt endpoints.

In host mode, the processor core sets DISNYET_R (bit 12) to disable the sending of NYET handshakes. When set, all successfully received RX packets are acknowledged including the point at which the FIFO becomes full. This bit only has an effect in high-speed mode, where it is set for all interrupt transfers.

DMAREQ_ENA_R

In peripheral mode, the processor core sets DMAREQ_ENA_R (bit 13) to enable the DMA request for the RX endpoint.

In host mode, the processor core sets DMAREQ_ENA_R (bit 13) to enable the DMA request for the RX endpoint.

ISO_R / AUTOREQ_RH

In peripheral mode, the processor core sets ISO_R (bit 14) to enable the RX endpoint for isochronous transfers, and clears it to enable the RX endpoint for bulk or interrupt transfers.

In host mode, if the processor core sets AUTOREQ_RH (bit 14), the REQPKT_H bit automatically is set when the RXPKTRDY bit is cleared.

AUTOCLEAR_R

In peripheral mode, if the processor core sets AUTOCLEAR_R (bit 15), the RXPKTRDY bit automatically is cleared when a packet of USB_RX_MAX_PACKET bytes is unloaded from the RX FIFO. When packets of less than the maximum packet size are unloaded, RXPKTRDY must be cleared manually.

In host mode, if the processor core sets AUTOCLEAR_R (bit 15), the RXPKTRDY bit automatically is cleared when a packet of USB_RX_MAX_PACKET bytes is unloaded from the RX FIFO. When packets of less than the maximum packet size are unloaded, RXPKTRDY must be cleared manually.

USB Count 0 (USB_COUNT0) Register

The USB_COUNT0 register (see Figure 26-52) indicates the number of received data bytes in the endpoint 0 FIFO. The value returned changes as the contents of the FIFO change and is only valid while RXPKTRDY is set.

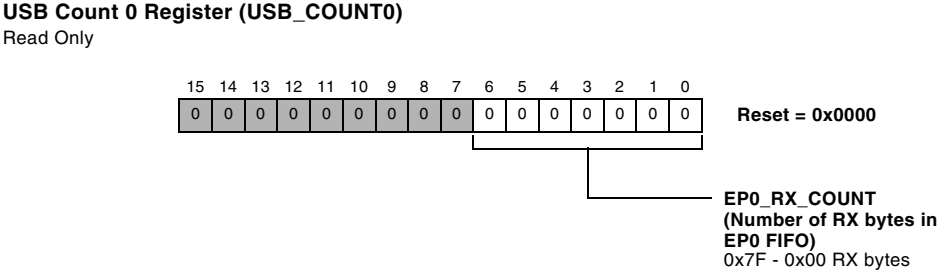


Figure 26-52. USB Count 0 Register

USB RX Byte Count EPx (USB_RXCOUNT) Register

The USB_RXCOUNT register (see Figure 26-53) holds the number of received data bytes in the packet in the RX FIFO. Note that the value returned changes as the FIFO is unloaded and is only valid while RXPKTRDY in USB_RXCSR is set.

USB RX Byte Count EPx Register (USB_RXCOUNT)
Read Only

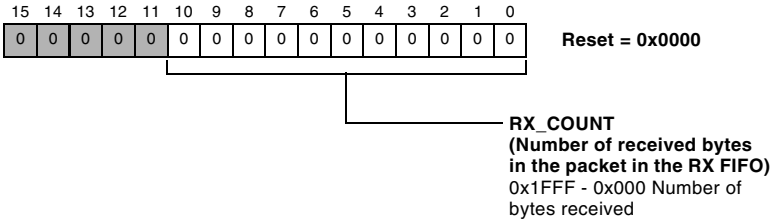


Figure 26-53. USB RX Byte Count Register

USB TX Type (USB_TXTYPE) Register

The USB_TXTYPE register (see Figure 26-54) selects the endpoint number and transaction protocol to use for the currently selected TX endpoint. There is a USB_TXTYPE register for each TX endpoint.

USB TX Type Register (USB_TXTYPE)
Read/Write

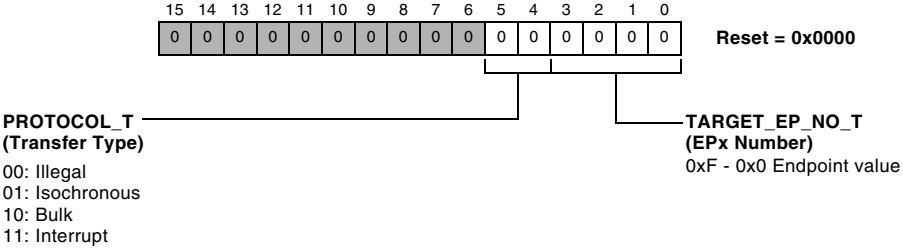


Figure 26-54. USB TX Type Register

USB NAK Limit 0 (USB_NAKLIMIT0) Register

The USB_NAKLIMIT0 register (see Figure 26-55) determines the number of frames/micro-frames after which the endpoint should timeout on receiving a stream of NAK responses for bulk endpoints.

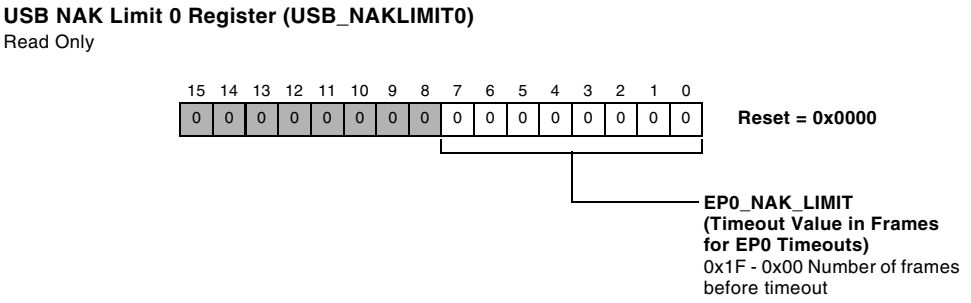


Figure 26-55. USB NAK Limit 0 Register

USB TX Interval (USB_TXINTERVAL) Register

The USB_TXINTERVAL register (see Figure 26-56) defines the polling interval for the currently selected TX endpoint for interrupt, isochronous, and bulk transfers. There is a USB_TXINTERVAL register for each configured TX endpoint, *except* endpoint 0

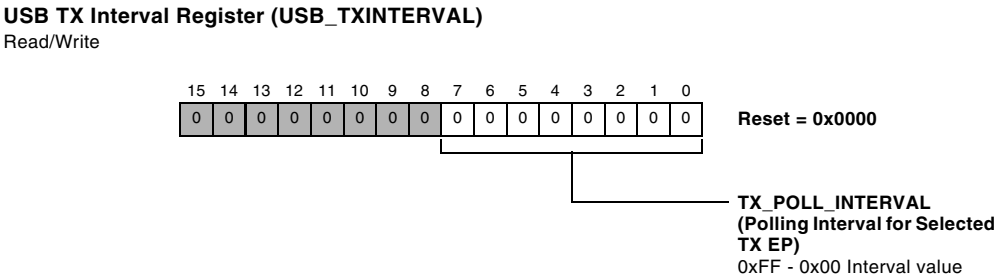


Figure 26-56. USB TX Interval Register

Table 26-4 relates transfer types to TX_POLL_INTERVAL values (number of frames).

Table 26-3. Interval Value Versus Transfer Type

Transfer Type	Speed	Valid Values (m)	Interpretation
Interrupt	Low Speed or Full Speed	1 – 255	Polling interval is m frames.
	High Speed	1 – 16	Polling interval is $2^{(m-1)}$ micro-frames.
Isochronous	Full Speed or High Speed	1 – 16	Polling interval is $2^{(m-1)}$ frames or micro-frames.
Bulk	Full Speed or High Speed	2 – 16	NAK Limit is $2^{(m-1)}$ frames or micro-frames. Note: A value of 0 or 1 disables the NAK timeout function.

USB RX Type (USB_RXTYPE) Register

The USB_RXTYPE register (see Figure 26-57) selects the endpoint number and transaction protocol to use for the currently selected RX endpoint. There is a USB_RXTYPE register for each RX, *except* endpoint 0.

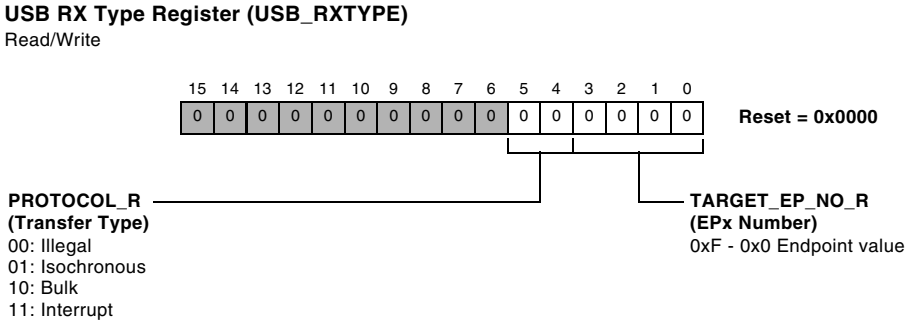


Figure 26-57. USB RX Type Register

USB RX Interval (USB_RXINTERVAL) Register

The USB_RXINTERVAL register (see Figure 26-58) defines the polling interval in number of frames for the currently selected RX endpoint for interrupt, isochronous, and bulk transfers. There is a USB_RXINTERVAL register for each configured RX endpoint, *except* endpoint 0.

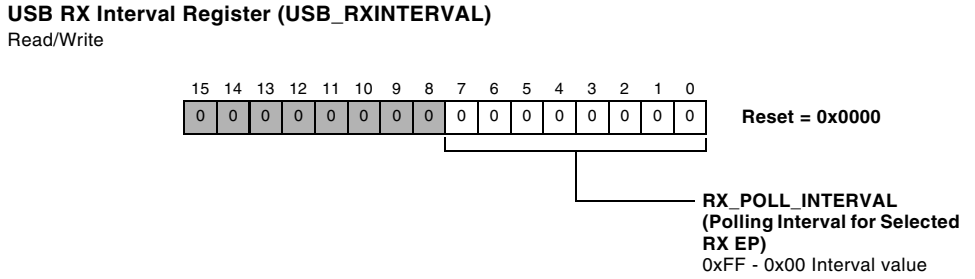


Figure 26-58. USB RX Interval Register

Table 26-4 relates transfer types to RX_POLL_INTERVAL values (number of frames).

Table 26-4. Interval Value Versus Transfer Type

Transfer Type	Speed	Valid Values (m)	Interpretation
Interrupt	Low Speed or Full Speed	1 – 255	Polling interval is m frames.
	High Speed	1 – 16	Polling interval is $2^{(m-1)}$ micro-frames.
Isochronous	Full Speed or High Speed	1 – 16	Polling interval is $2^{(m-1)}$ frames or micro-frames.
Bulk	Full Speed or High Speed	2 – 16	NAK Limit is $2^{(m-1)}$ frames or micro-frames. Note: A value of 0 or 1 disables the NAK timeout function.

USB TX Byte Count EPx (USB_TXCOUNT) Register

The USB_TXCOUNT register (see Figure 26-59) selects the size in bytes of the packet/transfer which is about to be written into a TX endpoint FIFO.

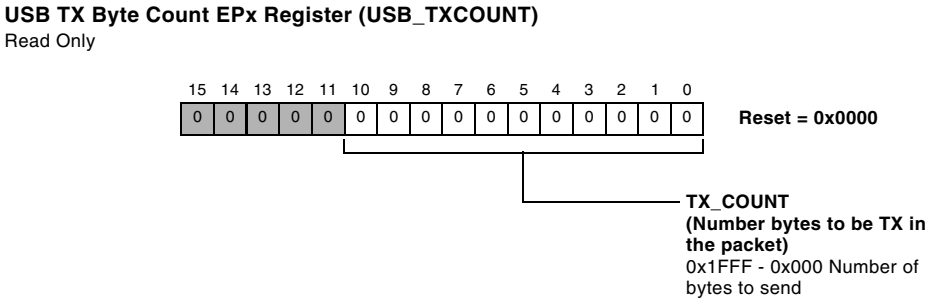


Figure 26-59. USB TX Byte Count EPx Register

As the packet is transferred, USB_TXCOUNT is a register that can be used by the processor core to program the size in bytes of the packet/transfer that is about to be written into a TX endpoint FIFO. The value is decremented by two when the processor core writes to the corresponding USB_EPx_FIFO high word address and is decremented by one when the processor core writes a byte to the FIFO using the corresponding USB_EPx_FIFO low word address. If the count itself reaches 0x0001 (which would only happen for odd-sized transfers), the next write into *either* USB_EPx_FIFO high word *or* USB_EPx_FIFO low word writes only the least significant byte of the half word into the FIFO. This aids DMA transfers that require IO accesses to go to the same address. USB_TXCOUNT must be re-loaded after it has counted to zero. It is not activated until it is loaded with a non-zero value.

See "Loading/Unloading Packets from Endpoints" on page 26-88 for more information about using USB_TXCOUNT.

USB Endpoint FIFO (USB_EPx_FIFO) Registers

Each endpoint uses a FIFO register (USB_EPx_FIFO) for data transfer. For more information about these FIFOs, see "Data Transfer" on page 26-88.

USB OTG Device Control (USB_OTG_DEV_CTL) Register

The USB_OTG_DEV_CTL register (see Figure 26-60) selects whether the USB controller is operating in peripheral mode or in host mode, and for controlling and monitoring the USB VBUS line.

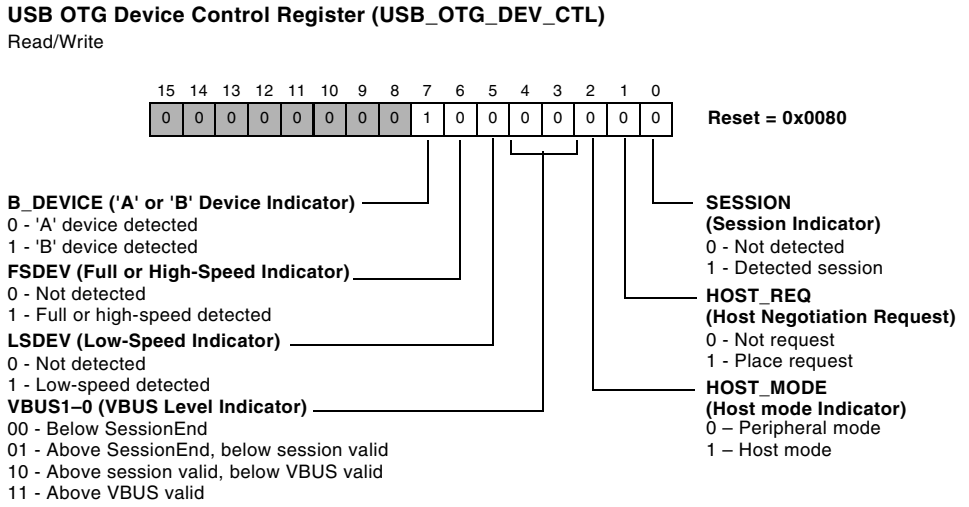


Figure 26-60. USB OTG Device Control Register

SESSION

When operating as an 'A' device, SESSION (bit 0) is set or cleared by the processor core to start or end a session. When operating as a 'B' device, SESSION is set/cleared by the USB controller when a session starts/ends.

SESSION is also set by the processor core to initiate the session request protocol. When the USB controller is in Suspend mode, the bit may be cleared by the processor core to perform a software disconnect.

HOST_REQ

When HOST_REQ (bit 1) is set, the USB controller initiates the host negotiation when suspend mode is entered. HOST_REQ is cleared when host negotiation is completed. ('B' device only)

HOST_MODE

The HOST_MODE (bit 2) read-only bit is set when the USB controller is acting as a host.

VBUS0[1:0]

The VBUS (bits 4–3) bits are read-only bits that encode the current VBUS level.

LSDEV

The LSDEV (bit 5) read-only bit is set when a low-speed device is detected being connected to the port. Only valid in host mode.

FSDEV

The FSDEV (bit 6) read-only bit is set when a full-speed or high-speed device is detected being connected to the port. High speed devices are distinguished from full-speed by checking for high-speed chirps when the device detects a USB reset. Only valid in host mode.

B_DEVICE

The B_DEVICE (bit 7) read-only bit indicates whether the USB controller is operating as the 'A' device or the 'B' device. Only valid while a session is in progress.

USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register

The USB_OTG_VBUS_IRQ register (see Figure 26-61) is an interrupt status register used to indicate when VBUS is required to be driven, charged or discharged as required by the OTG supplement. Writing a 1 to any of the bits 0 – 5 when they are active clears that bit and the corresponding interrupt. The USB_OTG_VBUS_IRQ register shares an interrupt source with USB_INTRUSB.

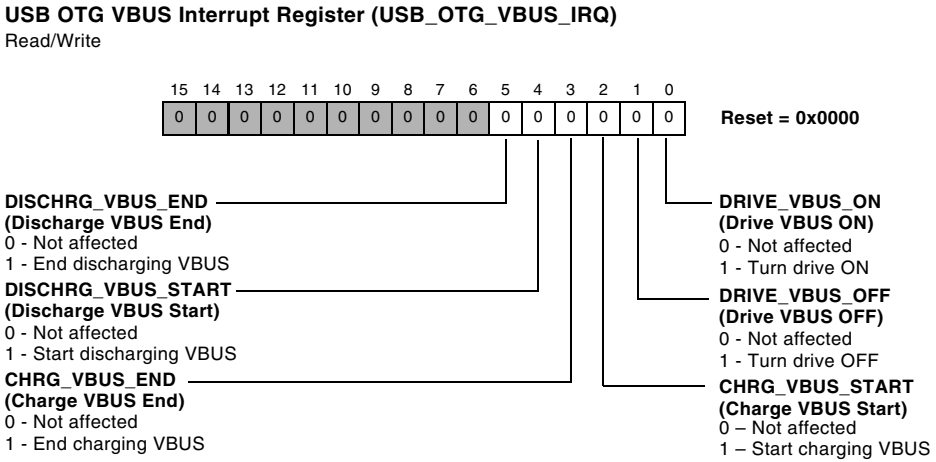


Figure 26-61. USB OTG VBUS Interrupt Register

Because the charge pump and VBUS charge/discharge circuit is located in a component/chip external to the on-chip PHY, the USB_OTG_VBUS_IRQ is provided as a means of allowing the software to drive the necessary control through a general-purpose, or dedicated IO.

DRIVE_VBUS_ON

When DRIVE_VBUS_ON (bit 0) is set, this status bit indicates the VBUS control circuit must be driven to greater than 4.4V ('A' device only).

DRIVE_VBUS_OFF

When DRIVE_VBUS_OFF (bit 1) is set, this status bit indicates the charge pump is to be shut off to end driving VBUS. ('A' device only).

CHRG_VBUS_START

When CHRG_VBUS_START (bit 2) is set, this status bit indicates the external control circuit is to begin charging VBUS to signal SRP ('B' device only).

CHRG_VBUS_END

When CHRG_VBUS_END (bit 3) is set, this status bit indicates the external VBUS control is to end charging of VBUS ('B' device only).

DISCHRG_VBUS_START

When DISCHRG_VBUS_START (bit 4) is set, this status bit indicates that VBUS is to be discharged in order to speed up VBUS discharging below SessionEnd threshold ('B' device only).

DISCHRG_VBUS_END

When DISCHRG_VBUS_END (bit 5) is set, this status bit indicates that VBUS control is to end discharging of VBUS ('B' device only).

USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register

The USB_OTG_VBUS_MASK register (see Figure 26-62) provides interrupt enable bits for the interrupt sources in USB_OTG_VBUS_IRQ.

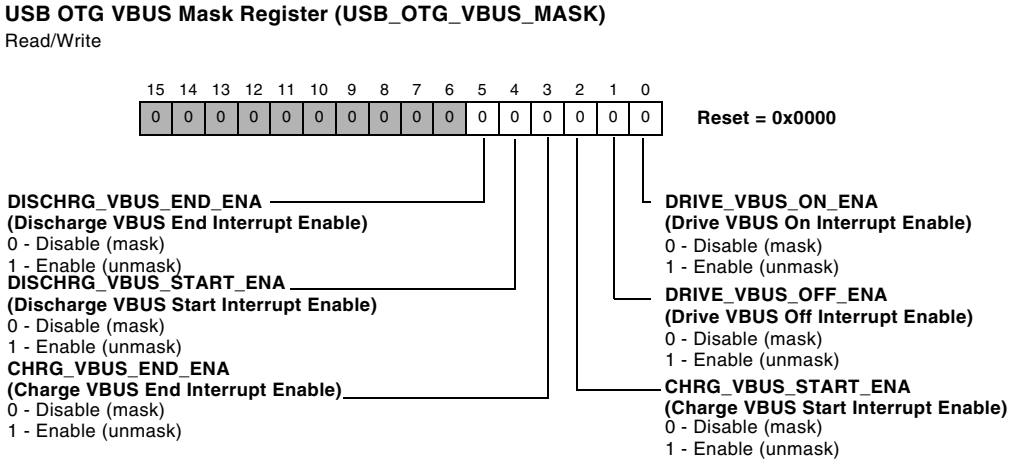


Figure 26-62. USB OTG VBUS Mask Register

USB Link Info (USB_LINKINFO) Register

The `USB_LINKINFO` register (see Figure 26-63) specifies PHY delays.

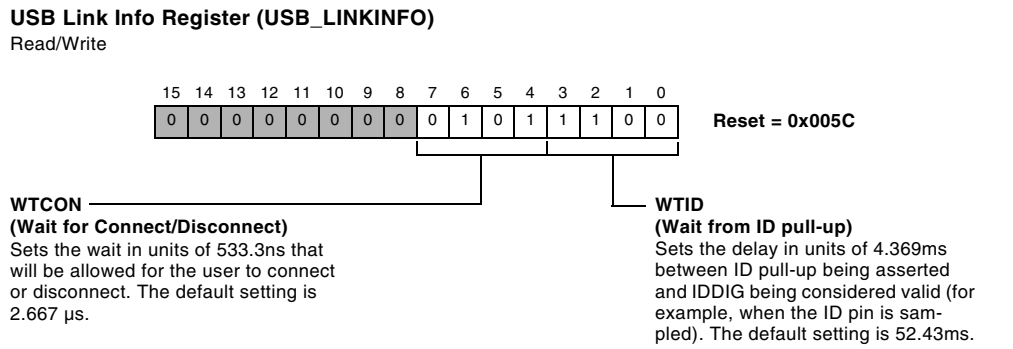


Figure 26-63. USB Link Info Register

USB VBUS Pulse Length (USB_VPLEN) Register

The USB_VPLEN register (see Figure 26-64) defines the duration of the VBUS pulsing charge for SRP initiation.

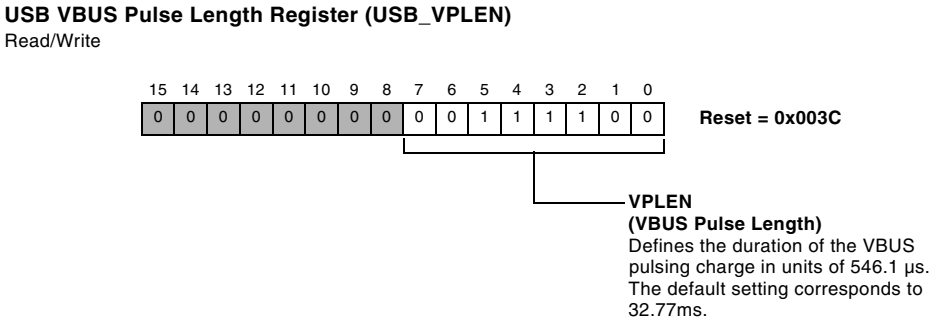


Figure 26-64. USB VBUS Pulse Length Register

USB High-Speed EOF 1 (USB_HS_EOF1) Register

For high-speed transactions, the USB_HS_EOF1 register (see Figure 26-65) defines the minimum time gap allowed between the start of the last transaction and the EOF.

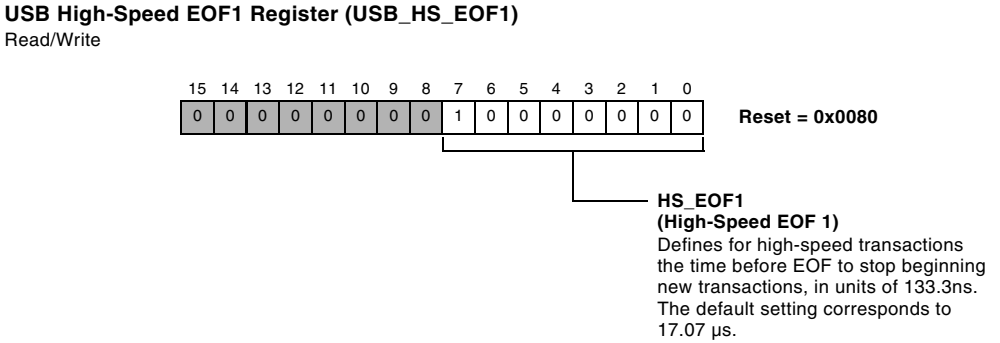


Figure 26-65. USB High-Speed EOF 1 Register

USB Full-Speed EOF 1 (USB_FS_EOF1) Register

For full-speed transactions, the USB_FS_EOF1 register (see Figure 26-66) defines the minimum time gap allowed between the start of the last transaction and the EOF.

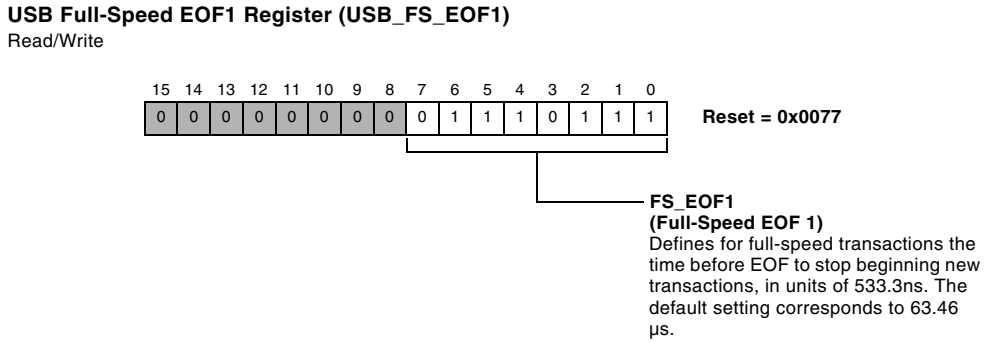


Figure 26-66. USB Full-Speed EOF 1 Register

USB Low-Speed EOF 1 (USB_LS_EOF1) Register

For low-speed transactions, the USB_LS_EOF1 register (see Figure 26-67) defines the minimum time gap allowed between the start of the last transaction and the EOF.

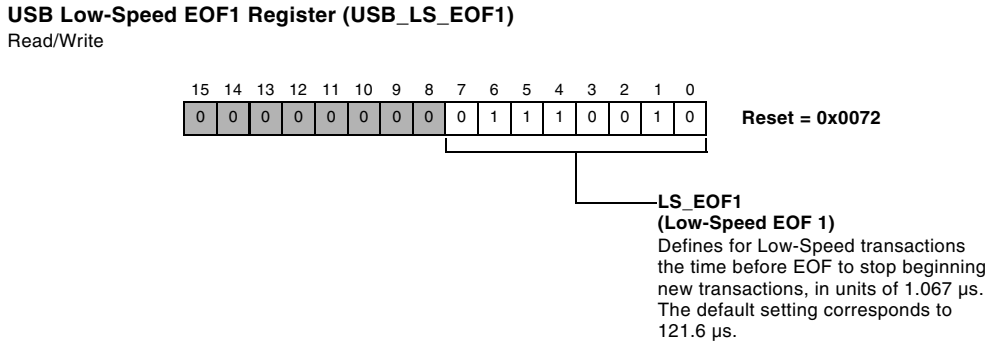


Figure 26-67. USB Low-Speed EOF 1 Register

USB APHY Control 2 (USB_APHY_CNTRL2) Register

The USB_APHY_CNTRL register (see Figure 26-68) controls transition of USB operation from normal to suspend to hibernate to resume normal.

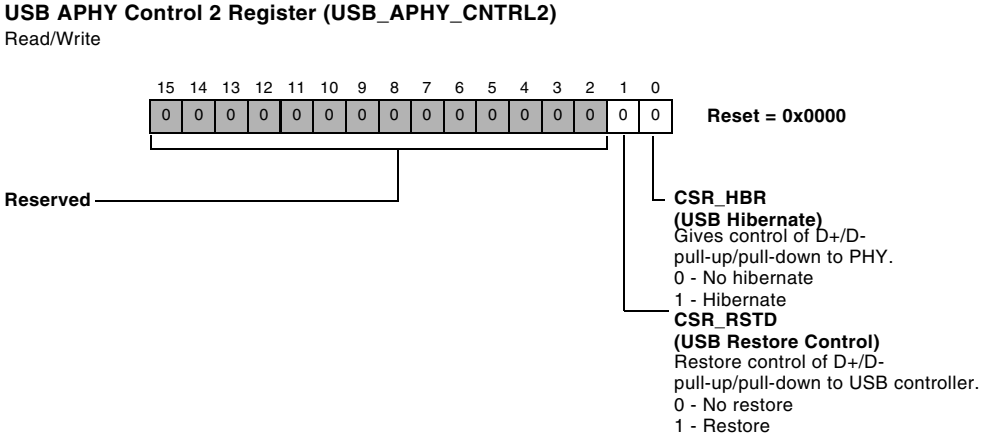


Figure 26-68. USB APHY Control 2 Register

CSR_HBR

The CSR_HBR (bit 0) signals the analog PHY to hold the state of the pull-up and pull-downs on the D+ and D– for hibernate.

CSR_RSTD

The CSR_RSTD (bit 1) signals the analog PHY to release its hold on the D+/D– pull-ups and pull-downs and give control back to the USB controller.

USB PLL OSC Control (USB_PLLOSC_CTRL) Register

The USB controller requires an internal clock of 960 MHz. This internal USB clock is generated from an external crystal. Recommended crystal values are 12 MHz, 24 MHz, and 30 MHz. Using a higher crystal frequency will result in less jitter. The USB PLL multiplier select field (USB_MSEL) should be programmed so that the resulting USB PLL output frequency will be 960 MHz, as defined by the following equation:

$$USB\ PLL\ output\ frequency = (2 \times USB\_MSEL \times CLKIN\_Freq)/(DF + 1)$$

where CLKIN_Freq is the value of the external crystal used. The reset value of this register clears the DF bit and sets the USB_MSEL to 20. Therefore, using a 24 MHz crystal will result in:

$$(2 \times 20 \times 24)/(0 + 1) = 960\ MHz$$

This register must be programmed before the USB module is enabled.

The USB_PLLOSC_CTRL register (see Figure 26-69) programs PLL and oscillator controls.

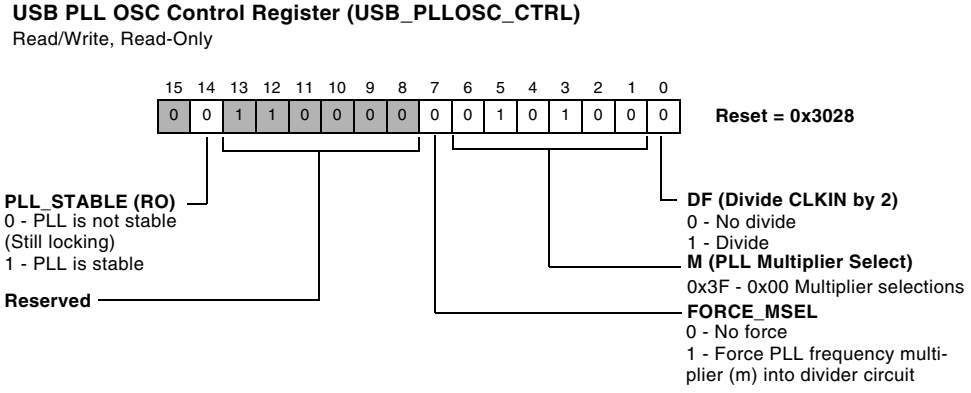


Figure 26-69. USB PLL OSC Control Register

USB SRP Clock Divider (USB_SRP_CLKDIV) Register

The USB_SRP_CLKDIV register (see Figure 26-70) programs the clock divider for sleep recovery of the USB peripheral (wakeup from sleep mode).

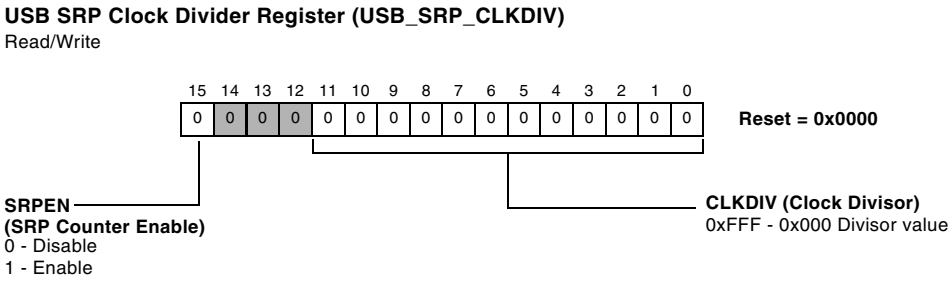


Figure 26-70. USB SRP Clock Divider Register

The processor is capable of running at peripheral clock frequencies up to 133 MHz. A 12-bit USB_SRP_CLKDIV register can be programmed to the desired value to divide the peripheral clock frequency that would clock the wakeup circuitry when the chip is put into sleep mode. For reliable operation of the circuit the user should program a value in the divider register that would divide the peripheral clock frequency greater than or equal to 32 kHz. The formula for calculating the value to be programmed into the USB_SRP_CLKDIV register is:

$$\frac{SCLK\ frequency\ in\ kHz}{32} - 1 = CLKDIV\ value$$

If SCLK = 130 MHz then CLKDIV = 4062 – 1 = 4061

If SCLK = 32 MHz then CLKDIV = 1000 – 1 = 999

USB DMA Interrupt (USB_DMA_INTERRUPT) Register

The USB_DMA_INTERRUPT register (see Figure 26-71) indicates which of the eight DMA master channels have a pending interrupt. The interrupt is generated when the corresponding DMA master channel DMA count register reaches zero. When the status is read by the processor core, software should write a 1 to the corresponding bit to clear the status.

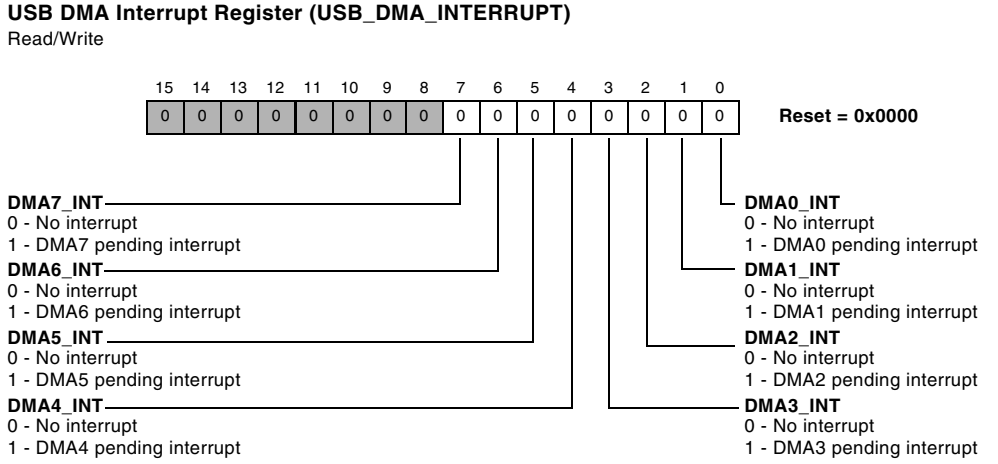


Figure 26-71. USB DMA Interrupt Register

USB DMAx Control (USB_DMA_CONTROL) Registers

There is one USB_DMAx_CONTROL register (see Figure 26-72 on page 26-145) for each DMA master channel. DMA control is used to assign, configure and control each endpoint with a corresponding DMA master channel. The n in the address below indicates the channel number 0 – 7.

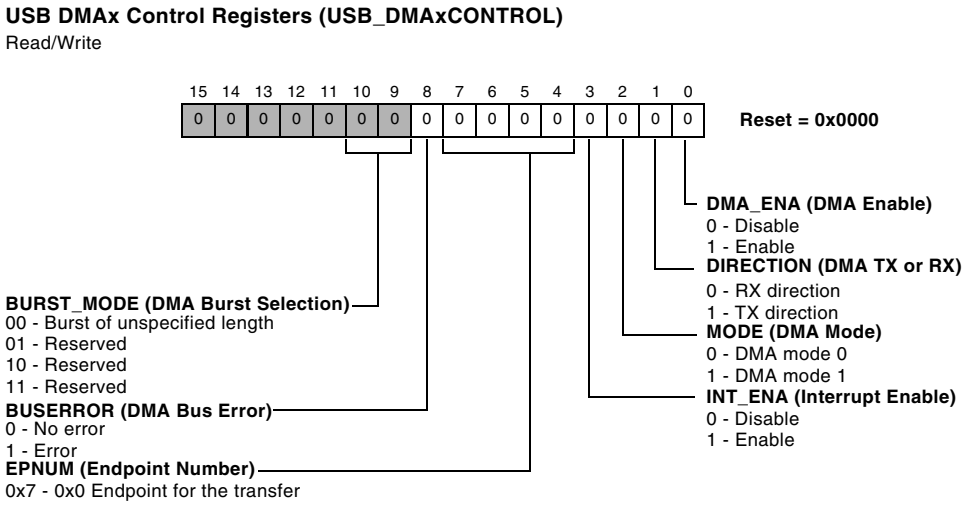


Figure 26-72. USB DMAx Control Registers

DMA_ENA

DMA_ENA (bit 0) enables the corresponding DMA master channel to allow it to transfer data between the FIFOs and on-chip memory.

DIRECTION

DIRECTION (bit 1) determines the direction of the DMA transfer. A value of 0 indicates a DMA write (for use with RX endpoints), and a 1 indicates a DMA read (for use with TX endpoints).

MODE

MODE (bit 2) determines whether the channel operates in DMA mode 0 or DMA mode 1.

INT_ENA

INT_ENA (bit 3) enables DMA interrupts for that channel (enable bit for the corresponding bit in the USB_DMA_INTERRUPT register).

EPNUM

The EPNUM (bits 7–4) value indicates the endpoint that is to be used for the DMA transfer. The only values that are valid in this implementation are 0 through 7.

BUSERROR

BUSERROR (bit 8) indicates a peripheral bus error was encountered by the master channel.

BURST_MODE

BURST_MODE (bits 10–9) determine the type of burst transfer the corresponding DMA channel uses to transfer data.

USB DMAx Address Low (USB_DMAxADDRLOW) Registers

The USB_DMAxADDRLOW registers (see Figure 26-73) hold the least-significant half word of the full 32-bit DMA address. This indicates the location in on-chip memory where DMA data is written or read.

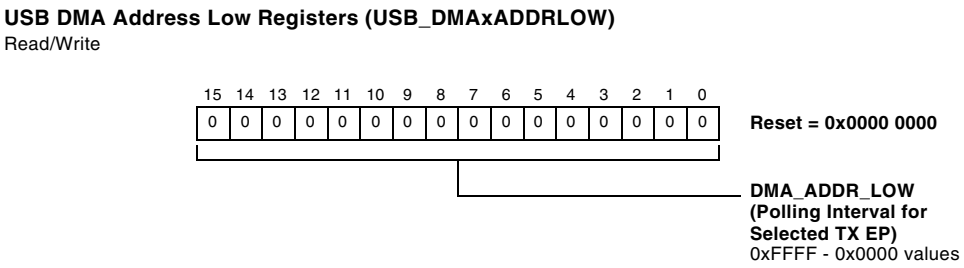


Figure 26-73. USB DMAx Address Low Registers

USB DMAx Address High (USB_DMAxADDRHIGH) Registers

The USB_DMAxADDRHIGH registers (see Figure 26-74) hold the most-significant half word of the full 32-bit DMA address. This indicates the location in on-chip memory where DMA data is written or read.

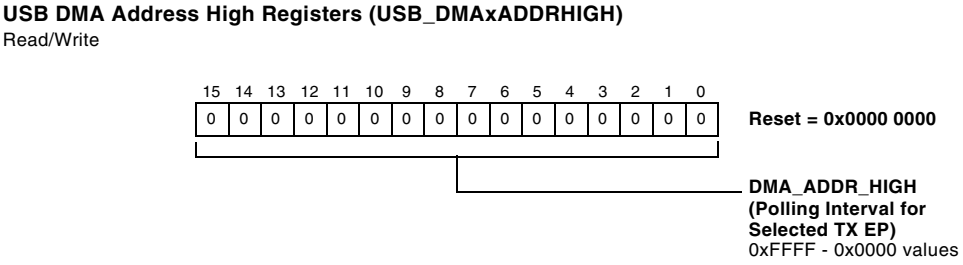


Figure 26-74. USB DMAx Address High Registers

USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers

The USB_DMAxCOUNTLOW registers (see Figure 26-75) hold the least-significant half word of the full 32-bit DMA count for each DMA channel. The 32-bit DMA count indicates the number of bytes to be transferred for a given DMA work block.

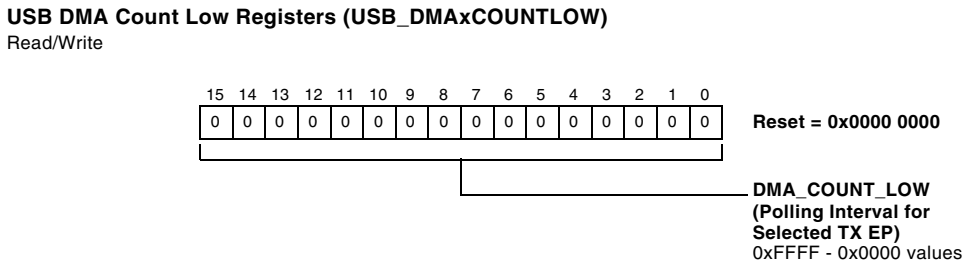


Figure 26-75. USB DMAx Count Low Registers

USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers

The USB_DMAxCOUNTHIGH registers (see Figure 26-76 on page 26-148) hold the most-significant half word of the full 32-bit DMA count for each DMA channel. The 32-bit DMA count indicates the number of bytes to be transferred for a given DMA work block.

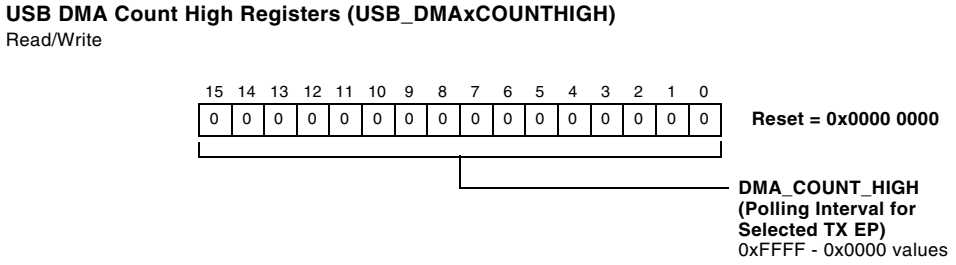


Figure 26-76. USB DMAx Count High Registers

References

The following references provide further information regarding the USB.

- *On-The-Go Supplement to the USB 2.0 Specification*, Rev 1.0a, June 24, 2003, USB-IF
- *Universal Serial Bus Specification 2.0*

Glossary of USB Terms

A list of common USB terms and their definitions as used in this specification and with respect to the USB controller follows:

'A' Device

The USB device with a mini-A plug inserted into its receptacle. The 'A' device always supplies power to VBUS.

'B' Device

The USB device with a standard-B or mini-B plug inserted into its receptacle. The 'B' device starts a session as the peripheral.

Bidirectional endpoint

An endpoint that can concurrently support receive and transfer packets.

Control endpoint

An endpoint that is solely used for transfer of USB control packets for setup and configuration. In all USB devices, the control endpoint refers to the bidirectional endpoint 0.

Dual role device

A USB device that can operate either as the USB host in an OTG session or as a traditional USB peripheral.

Endpoint

A single physical communication channel for USB, implemented as a FIFO and control logic for that endpoint. Each endpoint has an associated USB transfer type, maximum packet size, bandwidth requirement, endpoint number, and (often) a fixed transfer direction.

Frame

A regular, fixed 1ms time slot that can contain several transactions. The transfer type determines what transactions are permitted for a given endpoint.

HNP

Host negotiation protocol. Part of the USB OTG Supplement that allows the host function to be transferred between two connected dual role devices.

Packet

The lowest level of data exchange on USB. The size is determined by the transfer type and buffer size of the USB peripheral.

PHY

The PHY is a transceiver circuit that implements the physical layer of USB. For full speed USB OTG this includes line drivers and receivers, pull-up/pull-down resistors as well as device ID and VBUS level detection.

Session

A period during which USB transfers take place within an OTG connection. This can be initiated by the 'A' device (by driving VBUS) or 'B' device (by initiating SRP). VBUS is powered during a session.

SRP

Session request protocol. Part of the USB OTG Supplement that allows a 'B' device to turn on VBUS and initiate a USB session.

Transaction

Collection of one or more packets in sequence

Transfer

Collection of one or more transfers in sequence

Unidirectional endpoint

Endpoint with its direction fixed in a single direction (for example, it can only receive packets from the USB) in both host and peripheral modes.

27 SECURE DIGITAL HOST

This chapter describes the ADSP-BF54x processor Blackfin processor secure digital host (SDH) interface and includes the following sections:

- "Overview" on page 27-1
- "Interface Overview" on page 27-2
- "Description of Operation" on page 27-5
- "Functional Description" on page 27-8
- "Programming Model" on page 27-31
- "SDH Registers" on page 27-52
- "Programming Examples" on page 27-74

Overview

The ADSP-BF54x processor Blackfin processors provide an SDH interface for multimedia Cards (MMC), secure digital memory cards (SD card), and secure digital input/output cards (SDIO). All of these cards use similar interface protocols. The main difference between MMC and SD support is the initialization sequence. The main difference between SD card and SDIO support is the use of interrupt and read wait signals for SDIO.

Features of the SDH interface include:

- Support for a single SD or SDIO card
- Support for one or more MMC cards
- Support for 1-bit and 4-bit MMC and SD modes (SPI mode is not supported)
- Programmable clock frequency generated from SCLK
- Card detection capabilities (insertion/removal)
- SDIO interrupt and read wait features
- High capacity card support, such as SDHC, implemented within software
- 512-byte transmit/receive FIFO
- Dedicated DMA channel with 32-bit DMA access bus

Interface Overview

The SDH interface handles the multimedia and secure digital card functions. This includes clock generation, power management, command transfer, and data transfer. The bus interface converts 16-bit PAB accesses to 32-bit register accesses to the memory mapped registers; and generates interrupt requests to the processor core and system. The SDH has two interrupt signals (IRQ0 and IRQ1), that are fed to the system interrupt controller (SIC), IRQ72 and IRQ73 respectively.

The SDH block has 22 individual status bits in the SDH_STATUS register that can be configured to generate an interrupt. The status bits can be mapped to either of the two interrupts fed to the system interrupt controller. This allows for greater flexibility in system configuration.

To generate an interrupt on IRQ0, the interrupt should be enabled by setting the corresponding bit in the SDH_MASK0 register. Interrupts that are required to be generated on IRQ1 are enabled by setting the corresponding bit in the SDH_MASK1 register.

In addition to the status flags in the SDH_STATUS register generating interrupts, each of the flags in the SDH_ESTAT register are also capable of generating an interrupt. Interrupts for the SDH_ESTAT flags are enabled by setting the corresponding bit in the SDH_EMASK register. The interrupts are sent to the SIC through IRQ0.

The 32-bit DAB bus allows for efficient transfer of data, both to and from internal memory, using a dedicated DMA channel.

The SDH is a 6-pin interface (see Figure 27-1 on page 27-4) consisting of:

- SD_CLK
  The clock signal applied to the card from the SDH. All command and data signal transfers are synchronous to this clock. The frequency can vary between zero and the maximum clock frequency. Refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* for maximum supported clock frequencies.

- SD_CMD
  A bidirectional command signal used for the transfer of commands and card initialization. Using this signal, the SDH sends commands to the card, and the card sends responses back to the SDH. This signal can be configured for both push-pull mode and open-drain mode, but only MMC cards support the open-drain mode. The open-drain mode allows for multiple MMC cards to share data and command signals on the SDH interface, and allows for the initialization sequence to take place on all cards.

- SD_DATA3-0
  These are the configurable bidirectional data channels used for all data transfers to and from the card. The data bus width can be configured as 1-bit or 4-bit. For specific commands, the device can also drive its status onto the data lines.

Although multiple MMC cards can be bused to the single SDH interface, an MMC card cannot be bused with an SD or SDIO card such that they share the command and/or data signals. Multiple MMC cards bused together respond to CMD1 and CMD2 commands simultaneously, using open drain drivers. For any other card type, broadcast commands with a response must not be issued if the command or data signals are shared between cards.

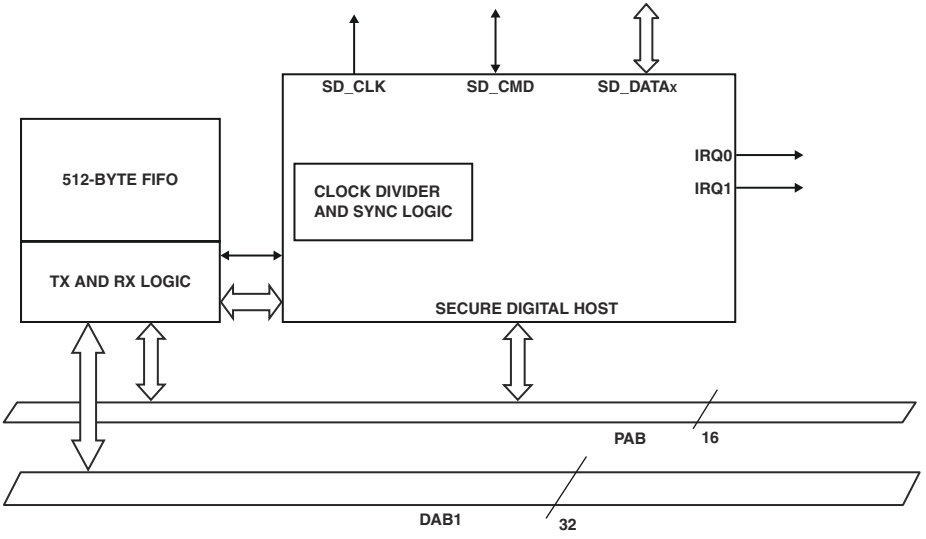


Figure 27-1. SDH Interface Block Diagram

Table 27-1 shows the functional operations for the SDH interface pins in all supported protocol modes.

Table 27-1. SDH Protocol Interface

Signal Name	MMC (1-bit)	MMC (4-bit)	SD (1-bit)	SD (4-bit)	SDIO (1-bit)	SDIO (4-bit)	Direction
SD_CLK	CLK	CLK	CLK	CLK	CLK	CLK	Output
SD_CMD	Com-mand/ response	Com-mand/ response	Com-mand/ response	Com-mand/ response	Command	Command	Bidirectional
SD_DATA 0	Dat0	Dat0	Dat0	Dat0	Dat0	Dat0	Bidirectional
SD_DATA 1	Not used	Dat1	Not used	Dat1	Interrupt	Dat1 or interrupt	Bidirectional
SD_DATA 2	Not used	Dat2	Not used	Dat2	Read wait	Dat2 or read wait	Bidirectional
SD_DATA 3	Not used /card detect	Dat3 /card detect	Not used /card detect	Dat3 /card detect	Not used	Dat3	Bidirectional

Description of Operation

The SDH controller is a fast, synchronous peripheral that uses various protocols to communicate with MMC, SD and SDIO cards. The SDH is compatible with the following protocols:

- MMC (Multimedia Card) bus protocol
- SD (Secure Digital) bus protocol
- SDIO (Secure Digital Input Output) bus protocol

The SDH does not support the SPI bus protocol.

Communication takes place through master and slave configurations where the SDH is the master and the card is the slave device. The SDH communicates with the card using a message-based bus protocol in which the host sends commands serially using the SD_CMD signal. Some commands require that the card provide a response back to the host. This response is also sent serially using the SD_CMD signal.

Data transfers, both to and from the card, occur using the SD_DATAx signals. The number of data lines used for the data transfer can be configured to either one or four using SD_DATA0 or SD_DATA3-0. All SD_CMD and SD_DATAx transfers are synchronous with SD_CLK.

Cyclic redundancy codes (CRC) are used to protect commands, responses and data transfers from transmission errors. Every command sent by the host and almost every response returned by the card on SD_CMD, generates a CRC7 code. A CRC16 code is used on the SD_DATAx signals to protect block data transfers. In the 4-bit bus configuration, CRC16 is calculated for each individual data signal.

When it is powered and detected or has been reset, a device connected to the SDH must be identified and initialized by the host. The software determines whether the device is compatible with the SDH controller and the implemented software drivers. This phase in the procedure is known as the card identification mode.

When a device is in card identification mode, the host performs the following actions:

- Reset the device
- Validate the device operating voltage range
- Identify the device type
- Assign/request a relative card address (RCA)

The card will only transition to a stand-by state when it has been assigned an RCA and it is known to be in data transfer mode. Data transfers can only take place when the device has entered the data transfer mode. All communications between the host and card during the card identification phase occur using the SD_CMD signal. The maximum clock frequency during this identification phase is typically far lower than the maximum data transfer frequency for the card.

Once the device is in data transfer mode, communication can take place using the SD_CMD and the SD_DATAx signals. The card is further interrogated to identify bus widths, maximum clock frequency, and the device capacity. At this point the bus width can be altered and the clock frequency from the SDH can be increased.

Data can be written to the card or read from the card using two different methods:

- Stream reads and writes
- Block reads and writes

Stream transfers produce a continuous stream of data until the SDH stops the transfer by setting a specific command. For stream read and write operations, additional maximum operating frequency limitations may be imposed by the device. Stream write operations may also have restrictions that are dependent upon writable block boundaries.

Block based transfers result in a block of a pre-configured size being transferred. The block size depends on the device and is obtained by reading registers contained on the device during the device detection procedure.

Functional Description

The following sections describe the functions and features of the SDH controller as well as the MMC, SD, and SDIO protocols. For further detailed information on timing parameters and protocol requirements, refer to *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet* and the following standards and specifications.

- MMCA System Specification
- JESD84 series of JEDEC standards
- SD Specifications Part 1 Physical Layer Specification
- SD Specifications Part 1 Physical Layer Simplified Specification
- SD Specifications Part E1 SDIO Specification

SDH Clock Configuration

The SDH is a fast, synchronous peripheral with a programmable clock frequency that is supplied by the SD_CLK signal. The interface between the SDH and the PAB/DAB busses operates at SCLK frequency. Communication between the clock domain that is supplied externally from the SDH on the SD_CLK signal and the internal SDH access to the PAB and DAB busses is accomplished using synchronizers in the SDH module. The SD_CLK frequency is configured by the 8-bit CLKDIV field and the CLKDIV_BYPASS bit of the SDH_CLK_CTL register (see "SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55.

If CLKDIV_BYPASS is set, the clock frequency driven on the SD_CLK signal is derived directly from SCLK.

If CLKDIV_BYPASS is cleared, the clock divider logic provides an SD_CLK frequency as shown below, where CLKDIV is an 8-bit value ranging between 0 and 255.

$$SD\_CLK = \frac{SCLK}{2 \times (CLKDIV + 1)}$$

The SD_CLK output is enabled or disabled by the CLK_E bit in the SDH_CLK_CTL register. A power save feature is implemented by setting PWR_SV_E, which disables the SD_CLK output when there are no transfers taking place on the SDH interface.

SDH Interface Configuration

The SDH supports multiple card types under various protocols. Different card types may require slightly different interface configurations.

The command signal on MMC cards operates in two different modes, depending upon the operating mode of the card. During the card identification mode, the command signal operates in open-drain configuration; but when the card enters data transfer mode, the signal is configured to push-pull mode.

The internal pull-up resistor for the SD_CMD signal is only intended to keep the signal from floating. The internal pull-up resistor is not sufficient during the card identification phase when the MMC card SD_CMD signal is operating in open-drain mode. If support for MMC devices is required, an external pull-up resistor should be added to the SD_CMD signal as detailed in the JEDEC standard.

The bus width used for data transfers is configurable to either 1-bit or 4-bits using the WIDE_BUS bit in the SDH_CLK_CTL register (see "SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55.

To stop signals from floating when no card is inserted or during times when all card drivers are in a high-impedance mode, various pull-up and pull-down resistor configurations can be enabled on the SDH_CMD and SDH_DATAx signals. The SDH_CFG register (see "SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55 provides the following options:

- Enable or disable pull-down on the SD_DATA3 signal
- Enable or disable pull-up on the SD_DATA3 signal
- Enable or disable pull-ups on the SD_DATA2-0 and SD_CMD signals

Card Detection

The SDH allows for software to detect when a card is inserted into its slot. SD and SDIO cards have an internal pull-up on the SD_DATA3 line that can be used as a card detect signal to indicate to the SDH that a card is present. After reset and once GPIO pins are configured for SDH functionality, the SD_DATA3 signal powers-up low due to a pull-down resistor that is enabled by default. When a card is inserted into the slot, a rising edge is detected on SD_DATA3 by the SDH and the SD_CARD_DET bit is set in the SDH_E_STATUS register. Once the card is detected the pull-down resistor on the SD_DATA3 signal should be disabled by clearing the PD_SDDAT3 bit of the SDH_CFG register; and the pull-up should be enabled by setting the PUP_SDDAT3 bit. Once the card has been correctly identified, the pull-up resistor within the card must also be disabled by issuing the SET_CLR_CARD_DETECT command.

Most SD/MMC sockets contain two additional signals for card detect and write protect functionality. It is highly recommended that card detection be implemented by using these signals. Figure 27-2 on page 27-12 shows a typical interface between the SDH interface and the card socket. The card detect signal should be debounced and interfaced to a GPIO signal. This provides the most robust and reliable method of card detection and is compatible with all SD/SDIO and MMC devices. In addition to providing card detect functionality, it also allows for interrupt driven card removal detection.

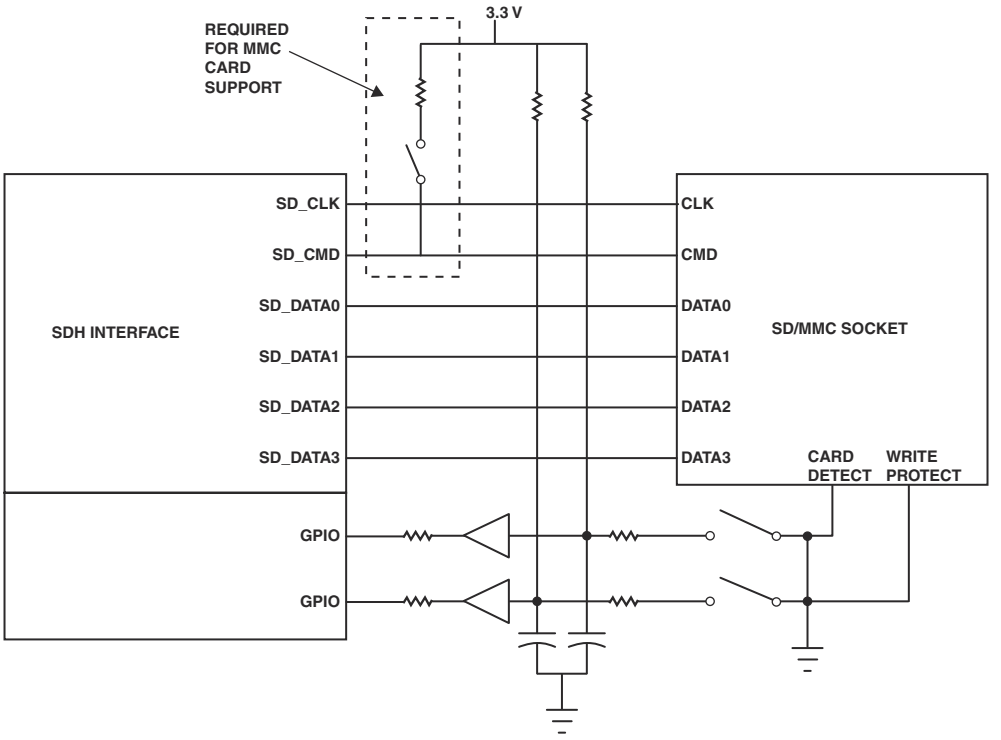


Figure 27-2. Example SD/MMC Socket interface to SDH

SDH Power Saving Configuration

The SDH requires two internal clock signals that are derived directly from SCLK. One of these clock signals is routed to the clock divider and generates the SD_CLK clock.

For the SDH to function, these clocks must be enabled by setting CLKS_EN in the SDH_CFG register. Clearing CLKS_EN disables the SDH regardless of other SDH clock configurations. The SD_CLK signal can be enabled or disabled using CLK_E in the SDH_CLK_CTL register.

Additional power saving options can be implemented by setting PWR_SV_E, which disables the SD_CLK output when there are no transfers taking place on the SDH interface. These configurations are shown in Table 27-2

Table 27-2. SDH Power Saving Configurations

CLKS_EN	CLK_EN	PWR_SV_E	SDH State	SD_CLK output
0	0	0	Disabled	No clock
0	0	1	Disabled	No clock
0	1	0	Disabled	No clock
0	1	1	Disabled	No clock
1	0	0	Enabled	No clock
1	0	1	Enabled	No clock
1	1	0	Enabled	Continuous clock[1]
1	1	1	Enabled	Clock only driven during transfers[1]

1 The PWR_ON field of the SDH_PWR_CTL register must be set to 0x3. If PWR_ON is 0x0—the clock will not output.

SDH Commands and Responses

The SDH sends commands to and receives responses from the card using the SD_CMD signal. A command to be sent to the card is issued by writing to the SDH_COMMAND register (see "SDH Command Register (SDH_COMMAND)" on page 27-57). This register contains a 6-bit CMD_IDX field that contains the command index to be sent to the card. The command index provides support for a total of 64 commands—0 (CMD0) to 63 (CMD63). Some commands require that an argument be sent with the command, an address for a read transaction for example. An argument is always sent with the command and it is the responsibility of the card to either ignore or use the argument field based on the command that is received. The argument sent with the command is provided using the SDH_ARGUMENT register (see "SDH Argument Register (SDH_ARGUMENT)" on page 27-57).

All command transfers are protected by a 7-bit cyclic redundancy check (CRC) code, more commonly referred to as a CRC7 checksum. This allows for transmission errors to be detected and for the command to be re-issued to the card in the event of an error. All commands sent to the card are composed of 48-bits as shown in Table 27-3 on page 27-14.

Table 27-3. SDH Command Format

Bit Position	Width	Value	Description
47	1	0	Start bit
46	1	1	Transmitter bit
[45:40]	6	-	Command index
[39:8]	32	-	Argument
[7:1]	7	-	CRC7 checksum
0	1	1	End bit

The CMD_RSP and CMD_L_RSP fields of the SDH_COMMAND register also provide configuration information about whether a response is to be expected back from the card and the type of response. The SDH can be configured for any of the following responses.

- No response
- Short response (see Table 27-4 on page 27-15)
- Long response (see Table 27-5 on page 27-16)

Table 27-4. SDH Short Response Format

Bit Position	Width	Value	Description
47	1	0	Start bit
46	1	0	Transmitter bit
[45:40]	6		Command index or check bits[1]
[39:8]	32		Card status, register contents or argument field
[7:1]	7		CRC7 checksum or check bits[2]
0	1	1	End bit

1 Responses that do not contain the command index have b#111111 in the check bits field.
2 Responses that do not contain a CRC7 check sum have b#111111 in the check bits field.

Table 27-5. SDH Long Response Format

Bit Position	Width	Value	Description
135	1	0	Start bit
134	1	0	Transmitter bit
[133:128]	6	111111	Check bits[1]
[127:1]	127	-	Register contents including internal CRC7[2]
0	1	1	End bit

1 Responses that do not contain the command index have b#111111 in the check bits field.
2 Responses that do not contain a CRC7 check sum have b#111111 in the check bits field.

Like the commands, all responses are sent on the SD_CMD signal. A response always has a '0' start bit followed by a '0' transmission bit to indicate the transfer is from card to SDH. Unlike the commands issued by the SDH, not all responses are protected by a CRC7 checksum. Refer to the appropriate specification for full details on the response formats for a specific device and whether they are protected by a CRC7 checksum.

When a short response is received, the 48-bit response is broken down by the SDH. The 32-bit field containing bits[39:8] is stored to SDH_RESPONSE0, where bit 39 of the response corresponds to bit 31 of SDH_RESPONSE0; and bit 8 corresponds to bit 0 of SDH_RESPONSE0. Bits[45:40] of the response are stored to the RESP_CMD field of the SDH_RESP_CMD register.

For a long response, bits [127:1] of the response are stored in SDH_RESPONE3-0. Bit 31 of SDH_RESPONSE0 contains the most significant bit (bit 127) of the response, and bit 1 of SDH_RESPONSE3 contains bit 1 of the response. Bit 0 of SDH_RESPONSE0 is always zero.

Figure 27-3 on page 27-18 shows the command path state machine. For the state machine to be active, the SDH must be enabled through CLKS_EN. Disabling the clocks to the SDH results in the state machine returning to the IDLE state.

The command path state machine is responsible for setting and clearing a number of status flags in the SDH_STATUS register (see "SDH Status Register (SDH_STATUS)" on page 27-63). Table 20-6 lists the status flags that are affected by the command path state machine.

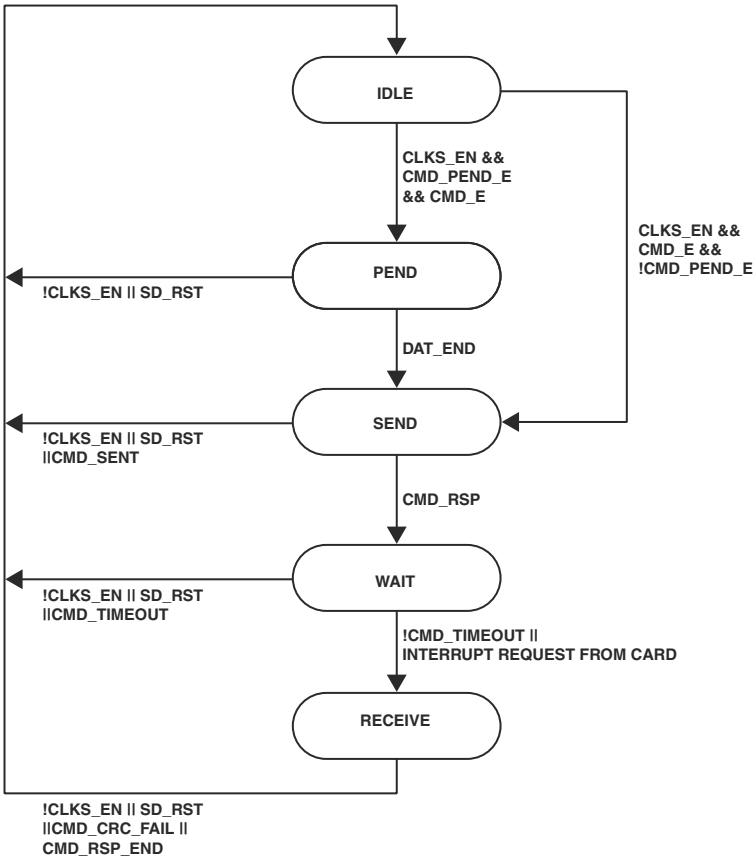


Figure 27-3. Command Path State Machine

Table 27-6. SDH Command Path Status Flags

SDH_STATUS Flag	Description	State Flag Set in
CMD_ACT	Command transfer is in progress	WAIT_S
CMD_SENT	Command without response sent successfully	SEND
CMD_TIMEOUT	Response timeout occurred (64 SD_CLK cycles)	WAIT_S
CMD_CRC_FAIL	Response CRC failure	RECEIVE
CMD_RSP_END	Response CRC successful	RECEIVE

The command path operates in a half-duplex mode, so that commands and responses can either be sent or received. If the state machine is not in the SEND state, the SD_CMD output is in high impedance state.

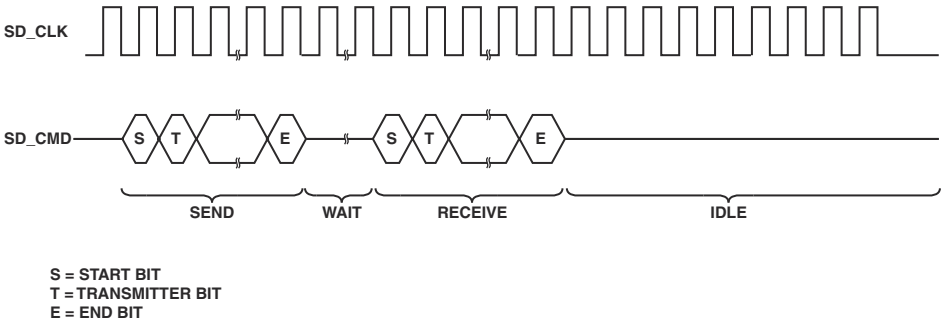


Figure 27-4. SDH Command Transfer

The following sections describe the SDH command path states.

IDLE State

The command path state machine remains in the IDLE state when it is not active. The command path state machine is enabled and leaves the IDLE state when the CMD_EN bit of the SDH_COMMAND register is set. The state goes to the PEND state if the CMD_PEND_E bit in SDH_COMMAND register is set; otherwise it enters the SEND state.

When the command path state machine returns to the IDLE state from another state and the return is not due to SDH being disabled or reset—the state machine remains in the IDLE state for at least eight SD_CLK cycles. During this time, the SDH continues to drive the SD_CLK signal even if the PWR_SV_E feature is enabled. This allows the card to complete the current operation. If enabled again, the state machine leaves the IDLE state only after the eight SD_CLK cycles have passed.

PEND State

The SDH enters the PEND state if the CMD_PEND_E bit in the SDH_COMMAND register is set. The state machine remains in the PEND state until it is notified by the data path sub block that the data transfer has completed. This is indicated by the DAT_END flag being set when the SDH_DATA_CNT decrements to zero. This mode allows for the automatic transmission of a STOP_TRANSMISSION command after reading or writing the required amount of data for stream-based transactions.

The CMD_PEND_E feature is not functional for block-based transfers and cannot be used to automatically issue the STOP_TRANSMISSION command for MULTIPLE_BLOCK_READ or MULTIPLE_BLOCK_WRITE operations.

SEND State

During the SEND state, the SDH sets the CMD_ACT flag in the SDH_STATUS register to indicate a transfer is in progress. The behavior of the state machine after the command is sent, depends upon whether the command expects a response back from the card.

If no response is expected, the SDH clears the CMD_ACT flag and sets the CMD_SENT flag to indicate that a command operation without a response has been completed. The state then goes to IDLE.

If a response is expected, the SDH enters the WAIT state.

WAIT State

In the WAIT state, the SDH waits for a response to be received on the SD_CMD signal. Upon entering this state, an internal timer starts. If the response is not received within 64 SD_CLK cycles, the CMD_TIMEOUT flag is set and the CMD_ACT flag is cleared. The state machine then enters the IDLE state, awaiting the next action.

A response, sent back from the card and indicated by the "0" start bit on the SD_CMD signal, transitions the SDH to the RECEIVE state where it is ready to receive a short or long response.

The WAIT state can also detect card interrupts. This is an optional feature that applies only to MMC cards. The feature is enabled by setting the CMD_INT_E bit in the SDH_COMMAND register. When CMD_INT_E is set, the timeout timer that is normally started upon entry to the WAIT state is disabled. The SDH remains in this state until a card interrupt is detected. Cards that implement this feature may have functions with a delayed response that is triggered by an internal event in the card. Once the event is triggered the card sends the response. The SDH then detects this start bit of the response and proceeds to the RECEIVE state.

RECEIVE State

In the RECEIVE state the SDH reads the response on the SD_CMD signal from the card. Upon receiving either the short or long response, if the response passes the CRC check, the CMD_ACT flag is cleared and the CMD_RSP_END flag is set. If the CRC check fails, the CMD_CRC_FAIL flag is set. In either case, the state machine then goes to the IDLE state.

SDH Command Path CRC

For all commands and responses, the CRC generator calculates the 7-bit CRC checksum for the 40 bits preceding the CRC code on the 48-bit signal. This includes the start bit, transmitter bit, command index, and command argument (or card status). The 7-bit CRC checksum is calculated for the first 120 bits of the register contents field for the long response format. Note that the start bit, transmitter bit, and the six check bits are not used in the CRC calculation for the long response. The command and response CRC checksum is a 7-bit value calculated as follows:

$$CRC[6:0] = Remainder \frac{x_7 \times M(x)}{G(x)}$$

with:

$$G(x) = x_7 + x_3 + 1$$

and for a short response:

$$M(x) = x_{39} \times (start\ bit) + \ldots + x_0 \times (last\ bit\ before\ CRC)$$

or for a long response:

$$M(x) = x_{119} \times (start\ bit) + \ldots + x_0 \times (last\ bit\ before\ CRC)$$

SDH Data

Data transfers both to and from the SDH take place over the SDH data bus signals `SD_DATA[3:0]`. The SDH data bus width is configured by the `WIDE_BUS` field of the `SDH_CLK_CTL` register (see "SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55). The default is 1-bit bus mode, where data is transferred over the `SD_DATA0` signal.

Or 4-bit mode can be enabled, after first configuring the card. This mode transfers four bits per `SD_CLK` cycle using the `SD_DATA[3:0]` signals.

The SDH data path state machine operates at SD_CLK frequency. The state machine leaves the IDLE state when the DTX_E field of the SDH_DATA_CTL register is set, enabling the data transfer. The state entered upon leaving the IDLE state is determined by DTX_DIR. The data path state machine is shown in Figure 27-5 on page 27-23.

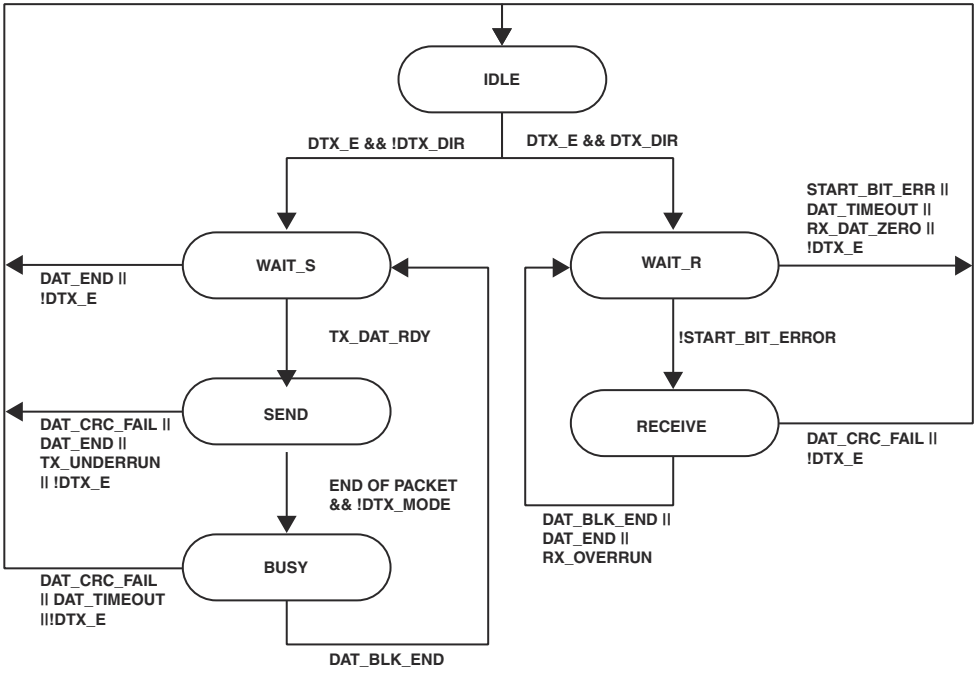


Figure 27-5. Data Path State Machine

The data path status flags are shown in Table 27-7.

Table 27-7. Data Path Status Flags

SDH_STATUS Flag	Description	States Flag Set in
TX_ACT	Data transmit in progress	WAIT_S
RX_ACT	Data receive in progress	WAIT_R
DAT_BLK_END	Data block sent successfully and CRC pass token received	BUSY (Block transfer mode only)
	Data block received correctly and CRC passed	RECEIVE (Block transfer only)
DAT_CRC_FAIL	Data block CRC failed on transmit	SEND (If transmitted data is not a multiple of DTX_BLK_LGTH.) BUSY (If CRC token indicates failure.)
	Data block CRC failed on receive	RECEIVE
DAT_TIMEOUT	Transmit timeout occurred before card de-asserted busy signal on SD_DATA0	BUSY
	Receive timeout error occurred before start bit of data detected	WAIT_R
DAT_END	All data sent	SEND
	All data received	RECEIVE
START_BIT_ERR	Start bit not detected on all SDH_DATAx signals	WAIT_R
TX_FIFO_STAT	Transmit FIFO is half empty	SEND
TX_FIFO_FULL	Transmit FIFO is full	SEND
TX_FIFO_EMPTY	Transmit FIFO is empty	SEND
TX_UNDERRUN	Transmit FIFO under run error	SEND
TX_DAT_RDY	Valid data available in the transmit FIFO	SEND
RX_FIFO_STAT	Receive FIFO is half empty	RECEIVE
RX_FIFO_FULL	Receive FIFO is full	RECEIVE

Table 27-7. Data Path Status Flags (Cont'd)

SDH_STATUS Flag	Description	States Flag Set in
RX_FIFO_EMPTY	Receive FIFO is empty	RECEIVE
RX_ OVERRUN	Receive FIFO over run error	RECEIVE
RX_FIFO_RDY	Valid data is available in the receive FIFO	RECEIVE

SDH Data Transmit Path

The transmit path consists of the WAIT_S, SEND and BUSY states. Both SDH_DATA_LGTH and SDH_DATA_TIMER must be configured before enabling the data path state machine with SDH_DATA_CTL. Upon leaving the IDLE state and entering the WAIT_S state, the SDH sets the TX_ACTIVE flag and copies SDH_DATA_LGTH into SDH_DATA_CNT.

The behavior of the SEND state depends on which transfer mode is configured.

- Stream Transfer Mode

  If the SDH is configured for stream transfer mode, the SDH sends data to the card until SDH_DATA_CNT expires, at which point the DAT_END flag is set and the state machine returns to the IDLE state.

  Additionally the transition of SDH_DATA_CNT to zero activates the command path state machine if it is currently in the PEND state.

  If at any point during the stream transfer the transmit FIFO becomes empty and data is not available in the FIFO by the time the next transfer is due to take place, the TX_UNDERRUN flag is set before returning to the IDLE state.

- Block Transfer Mode

  In block transfer mode, DTX_BLK_LGTH bytes, as specified during the write to SDH_DATA_CTL, are transmitted. Each byte transferred also decrements SDH_DATA_CNT.

  Upon completion of the block transfer, the SDH appends an internally generated 16-bit CRC code and an end bit to the data transferred over the SDH_DATAx signals. The SDH then waits for the card token response on the SD_DATA0 line to indicate whether the data was received correctly by the card or not.

  If the CRC response token sent by the card indicates the data was received correctly, the DAT_BLK_END flag is set before moving to the BUSY state. If the data was not received correctly, the DAT_CRC_FAIL flag is set before returning to the IDLE state.

  The decrementing of SDH_DATA_CNT to zero results in the DAT_END flag being set. If the total number of bytes transmitted for the current block results in the SDH_DATA_CNT decrementing to zero and the number of bytes transferred is not equal to DTX_BLK_LGTH, the transmission stops and the DAT_CRC_FAIL flag is set. The data path returns to the IDLE state.

  If at any point during the block transfer the transmit FIFO becomes empty and data is not available in the FIFO by the time the next transfer is due to take place, the TX_UNDERRUN flag is set before returning to the IDLE state.

During the BUSY state, SDH continuously samples SD_DATA0 which at this point is driven low by the card to indicate that the card is busy. When a logic high state is detected, indicating that the card is no longer busy, the state machine returns to the WAIT_S state. It then either returns to IDLE if all data has been sent, or moves back to the SEND state to start another block transfer.

Upon entering the BUSY state, the SDH started decrementing the time-out value specified by SDH_DATA_TIMER. If the SDH timeout counter expires before the SD_DATA0 signal is detected high, the SDH sets the DAT_TIMEOUT flag and returns to the IDLE state.

SDH Data Receive Path

The receive path consists of the WAIT_R and the RECEIVE states. Both SDH_DATA_LGTH and SDH_DATA_TIMER must be configured before enabling the data path state machine with SDH_DATA_CTL. Upon leaving the IDLE state and entering the WAIT_R state the SDH sets the RX_ACTIVE flag and copies SDH_DATA_LGTH into SDH_DATA_CNT. The behavior of the RECEIVE state is influenced by the transfer mode.

Once the receive path has entered the WAIT_R state after being enabled for a receive transaction, the SDH starts decrementing the timeout value supplied by the SDH_DATA_TIMER.

If the SDH is configured for a 1-bit data bus, the DAT_TIMEOUT flag is set if a start bit is not detected on the SD_DATA0 signal before the timeout counter reaches zero. The state machine then returns to the IDLE state.

If the SDH is configured for 4-bit bus mode and the start bit is not detected on all four SD_DATAx signals before the timeout counter expires—the DAT_TIMEOUT flag is set. The state machine returns to the IDLE state. If a start bit is detected on some, but not all, of the SD_DATAx signals on the same sampled SD_CLK cycle, then the START_BIT_ERR flag is set and the state machine returns to the IDLE state. Upon correct detection of the start bit, the state machine goes to the RECEIVE state.

The behavior of the RECEIVE state differs for stream and block transfers.

- For stream transfers, the received data is packed into bytes and written to the data FIFO. Data is continuously received and written to the data FIFO until SDH_DATA_CNT decrements to zero.

  When the counter reaches zero the remaining data in the shift register is written into the FIFO, the DAT_END flag is set and the state machine goes to the WAIT_R state.

  When the receive FIFO is detected empty, the RX_DAT_ZERO flag is set and the state goes to the IDLE state.

  If the data FIFO becomes full and data has not been read from the FIFO prior to the next byte being written to the FIFO, then the RX_OVERRRUN flag is set. The state goes to the WAIT_R state then into the IDLE state.

- In block transfer mode, the received data is packed into bytes and written to the data FIFO. When DTX_BLK_LGTH bytes have been received, the SDH reads the 16-bit CRC check bits. If the received CRC matches the internally calculated CRC, the DAT_BLK_END flag is set and the state goes to the WAIT_R state.

  If the SDH_DATA_CNT counter expires in alignment with the end of a DTX_BLK_LGTH block, the DAT_END and DAT_BLK_END flags are set, and the state goes to the WAIT_R state. When the receive FIFO is detected empty, the RX_DAT_ZERO flag is set, and the state goes to the IDLE state.

  If SDH_DATA_CNT expires before the end of a DTX_BLK_LGTH block, the DAT_CRC_FAIL flag is set. The state goes to the IDLE state.

SDH Data Path CRC

The data CRC generator of the SDH calculates the 16-bit CRC checksum for all bits sent or received for a given block transaction. The data path CRC generator is not enabled for stream based data transfers. For a 1-bit bus configuration, the 16-bit CRC is calculated for all data sent on the SD_DATA0 signal. For a 4-bit wide data bus, the 16-bit CRC is calculated separately for each SD_DATAx signal. The data path CRC checksum is a 16-bit value calculated as follows.

$$CRC[15:0] = Remainder\ \frac{x_{16} \times M(x)}{G(x)}$$

with:

$$G(x) = x_{16} + x_{12} + x_5 + 1$$

where:

$M(x) = x_{(8 \times DTX\_BLK\_LGTH)-1} \times$ *(first data bit)* $+ \ldots + x_0 \times$ *(last data bit)*

SDH Data FIFO

The data FIFO is a 32-bit wide, 16-word deep data buffer with transmit and receive logic. The FIFO configuration depends on the state of the TX_ACT and RX_ACT flags. If TX_ACT is set, the FIFO operates as a transmit FIFO, supplying data to the SDH for transfer to the card. If RX_ACT is set, the FIFO operates as a receive FIFO, where the SDH writes data received from the card. If neither TX_ACT nor RX_ACT flags are set, then the FIFO is disabled.

When the transmit FIFO is disabled, all the transmit status flags are de-asserted and the transmit read and write pointers are reset. The SDH asserts the TX_ACT flag upon starting a data transfer. During the data transfer the transmit logic maintains the transmit FIFO status flags shown in Table 27-8 on page 27-30.

Table 27-8. SDH Transmit FIFO Status Flags

SDH_STATUS Flag	Description
TX_FIFO_STAT	Transmit FIFO is half empty
TX_FIFO_FULL	Transmit FIFO is full
TX_FIFO_EMPTY	Transmit FIFO is empty
TX_UNDERRUN	Transmit FIFO under run error
TX_DAT_RDY	Valid data available in the transmit FIFO

When the receive FIFO is disabled, all the receive status flags are de-asserted and the receive read and write pointers are reset. The SDH asserts the RX_ACT flag upon starting a data read transaction. During the data transfer, the receive logic maintains the receive FIFO status flags shown in Table 27-9 on page 27-30.

Table 27-9. SDH Receive FIFO Status Flags

SDH_STATUS Flag	Description
RX_FIFO_STAT	Receive FIFO is half empty
RX_FIFO_FULL	Receive FIFO is full
RX_FIFO_EMPTY	Receive FIFO is empty
RX_OVERRUN	Receive FIFO under run error
RX_DAT_RDY	Valid data available in the receive FIFO

SDIO Interrupt and Read Wait Support

Two additional SDH features implement SDIO functionality.

- Hardware interrupt support over the SD_DATA1 pin
- Read wait request over the SD_DATA2 pin

SDIO devices can have multiple interrupt sources that are mapped to a single interrupt line. The interrupt is level sensitive, allowing multiple functions to generate an interrupt simultaneously. Thus the interrupt request will continually be asserted until all sources generating an interrupt are determined and cleared by the SDH.

The sources of the interrupts are found by interrogating the SDIO device. The interrupts are cleared through operations unique to each function.

The SDIO device sends an interrupt request to the SDH by asserting the SD_DATA1 signal low. The interrupt status is indicated by the SDIO_INT_DET bit of the SDH_E_STATUS register. The status can be configured to interrupt the processor through the SDIO_MSK bit of the SDH_E_MASK register.

When the SDH is configured for 1-bit bus width, the interrupt is generated by the SDIO with no timing constraints since the SD_DATA1 signal acts as a dedicated IRQ signal. The SDH should be configured using SDH_CFG such that pull-ups are enabled on all SD_DATAx signals. When the SDH samples SD_DATA1 low, the SDH asserts the SDIO_INT_DET flag. This flag is asserted until the SD_DATA1 signal is sampled high again.

When the SDH is configured for 4-bit bus width, the SD_DATA1 signal is shared between the IRQ signal and the SD_DATA1 signal. In this configuration the interrupt is only recognized by the SDH within a specific interrupt period.

Programming Model

This section contains the following procedures:

"Card Identification" on page 27-32

"Single Block Write Operations" on page 27-35

"Single Block Read Operations" on page 27-39

"Multiple Block Write Operations" on page 27-43

"Multiple Block Read Operations" on page 27-48

Card Identification

Before data transfers can take place between the SDH and the SD/MMC/SDIO device, the device type must be identified. During this phase, the SD_CLK frequency is typically limited to no more than 400 kHz.

SD Card Identification Procedure

Refer to the following documents for details on the SD commands and response types.

- *SD Specifications Part 1 Physical Layer Specification*
- *SD Specifications Part 1 Physical Layer Simplified Specification*

The SD card identification procedure is shown below.

1. Issue the IDLE command to the card using the SDH_COMMAND register.
2. Issue the SEND_IF command through the SDH_COMMAND register, supplying the SDH supported voltage and a check pattern using the SDH_ARGUMENT register. The command expects an R7 response type. If a valid response with a compatible voltage range and matching

check pattern is received, the card is compliant with SD veSDHon 2.00 or later. If a response is received with an incompatible voltage range the card cannot be used. If no response is received at all, as indicated by the CMD_RSP_TIMEOUT flag on the SDH_STATUS register, go to step 5.

3. Issue the SD_SEND_OP_COND command through the SDH_COMMAND register, supplying the voltage window supported and whether the host supports high capacity cards using the SDH_ARGUMENT register. The SDH expects an R3 response to this command. The SDH can reject the card at this point if the voltage range is not compatible.

   If the card returns a response indicating that it is busy, resend the SD_SEND_OP_CMD until the card indicates it is ready. If the host does not support the high capacity mode as indicated by setting the HCS bit of the argument to 0—a high capacity card never clears the busy status bit. The card should be identified within one second. If in that time the card is still busy, or no valid responses have been received, the card is rejected.

4. If the host supports high capacity cards, verify whether the response in the SDH_RESPONSEO register indicates that the card capacity status (CCS) bit is set. If CCS is set, an SD VeSDHon 2.00 or later high capacity SD memory card is present—proceed to step 6. If the CCS bit is cleared, then the card is an SD VeSDHon 2.00 or later standard capacity memory card—proceed to step 6.

5. Issue the SD_SEND_OP_COND command through the SDH_COMMAND register, supplying the voltage window supported and with the High Capacity Support (HCS) bit set to 0 using the SDH_ARGUMENT register. The SDH expects an R3 response to this command, at which point the card can be rejected if the voltage range is not compatible.

If the card returns a response indicating that it is busy, resend the SD_SEND_OP_CMD until the card indicates that it is ready. The card should be identified within one second. If in that time the card is still busy or no valid responses have been received, the card is rejected. Once the response indicates that the card is ready, the card type has been identified as an SD VeSDHon 1.x Standard Capacity memory card.

6. Issue the ALL_SEND_CID command through the SDH_COMMAND register. An R2 response type is expected. This results in the card sending contents of the 128-bit card identification (CID) register and transitioning from the ready mode to identification mode.

7. Issue the SEND_RELATIVE_ADDR command through the SDH_COMMAND register. An R1 response type is expected. This results in the card issuing a new relative address which must be used to select the card in the future for data transfers. The card then moves into standby mode, completing the identification procedure.

MMC Identification Procedure

1. Issue the IDLE command through the SDH_COMMAND register.

2. Issue the SEND_OP_COND command through the SDH_COMMAND register, supplying the operating voltage window that the host is compatible with and the access mode that the host supports, byte or sector, through the SDH_ARGUMENT register. The SDH expects an R3 type response. This allows the host to reject the card if it is not compatible with the supply voltage or if the access mode is not supported by the host software. If the card returns an indication that it is busy this step should be repeated until the card is either rejected or not busy.

3. Issue the ALL_SEND_CID command through the SDH_COMMAND register. The SDH expects an R2 response to this command. This results in the card sending the contents of the 128-bit card identification (CID) register and transitioning from ready to identification mode.

4. Issue the SET_RELATIVE_ADDR command through the SDH_COMMAND register, providing a 16-bit relative card address (RCA) through the SDH_ARGUMENT register that will get assigned to the card. An R1 response type is expected. This results in the card being assigned with the provided RCA which must be used to select the card in the future for data transfers. The card then moves into standby mode, completing the identification procedure.

Single Block Write Operations

Block write operations typically consist of 512 bytes of data per block. If the card is found to support other block lengths, or the default block length as specified in the CID register is not 512, then the block length of the SDH must be configured accordingly. The block length of the card and the block length of the SDH must be configured for the same block size at all times. The block length of the SDH is configured using the DTX_BLK_LGTH field of the SDH_DATA_CTL register.

It is important to pay attention as to when the data path state machine is enabled and when data is written to the FIFO for transfer to the card. Write transactions require that data be written after the response has completed for the WRITE_BLOCK command. If the data path state machine is enabled prior to sending the WRITE_BLOCK command, data must not be written to the transmit FIFO through the DMA or core until after the response has been received as indicated by the CMD_RSP_END event. Failure to adhere to this procedure can result in data being written to the card in violation to the block write timing parameters resulting in a data CRC failure.

Using Core

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.
3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.
5. Write to the SDH_DATA_LGTH register, the number of bytes to be transferred. This is 512 bytes for a single block.
6. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a write operation.
7. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card, and accesses are not enabled.
8. Write the WRITE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.
9. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, the clear the status bit using the SDH_STATUS_CLR register.

10. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E should also be set to enable the data path state machine. All other fields of the SDH_DATA_CTL register should be zero.

11. Write data to the SDH_FIFO register until the FIFO is full as indicated by the TX_FIFO_FULL flag of the SDH_STATUS register. Continue to write data to the FIFO as long as the FIFO is not full or write data in blocks of eight 32-bit words if polling on the TX_FIFO_STAT bit indicates the transmit FIFO is half empty. Continue to write data until all 128 32-bit words (512 bytes) have been transferred.

12. Wait for the DAT_BLK_END event that indicates the card has responded with the CRC token. If the SDH_DATA_LGTH register was set to 512 bytes in step 5, DAT_END is also set.

13. Clear the DAT_BLK_END and DAT_END flags in the SDH_STATUS_CLR register.

Using DMA

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.

2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.

3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.

4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.

5. Configure the DMA channel assigned to the SDH controller by writing the address of the first byte of data to be written to the card to the DMAx_START_ADDR register. The DMAx_X_COUNT register should be set to 128 and the DMAx_X_MODIFY register should be set to 4. The DMAx_CONFIG register should be set for DMA enable, a word size of 32-bits.

6. Once the DMA channel has been configured and enabled, write the number of bytes to be transferred to the SDH_DATA_LGTH register. This is 512 bytes for a single block.

7. Write the appropriate timeout value for a write operation to the SDH_DATA_TIMER register.

8. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card, and accesses are not enabled.

9. Write the WRITE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

10. Wait for the CMD_RSP_END event in the SDH_STATUS register. When the event is detected, clear the status bit using the SDH_STATUS_CLR register.

11. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E and DTX_DMA_E should also be set to enable the data path state machine and to allow the DMA controller to access the transmit FIFO. All other fields of the SDH_DATA_CTL register should be zero.

12. Wait for the DAT_BLK_END event that indicates the card has responded with the CRC token. If the SDH_DATA_LGTH register was set to 512 bytes in step 5, DAT_END is also set.

13. Clear the DAT_BLK_END and DAT_END flags in the SDH_STATUS_CLR register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register, if applicable.

Single Block Read Operations

Block read operations typically consist of 512 bytes of data per block. If the card supports other block lengths or the default block length as specified in the CID register is not 512—then the block length of the SDH must be configured accordingly. The block length of the card and the block length of the SDH must be configured for the same block size at all times. The block length of the SDH is configured using the DTX_BLK_LGTH field of the SDH_DATA_CTL register.

For data transfers from the card to the SDH, it is important to pay attention to when data is read from the receive FIFO relative to when the data path state machine is enabled. Read transactions can occur on the SD_DATAx signals prior to receiving the response of the command. Therefore the data path state machine, and the DMA controller if being used, should be enabled—either prior to issuing a command that involves a data read packet, or immediately after the command has been issued but before the CMD_RSP_END event occurs.

Using Core

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.
3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.
5. Write the number of bytes to be transferred to the SDH_DATA_LGTH register. This is 512 bytes for a single block.
6. Write the appropriate timeout value for a read operation to the SDH_DATA_TIMER register.
7. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card, and accesses are not enabled.
8. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E and DTX_DIR should also be set to enable the data path state machine and indicate that the transfer direction is from card to controller. All other fields of the SDH_DATA_CTL register should be zero.

9. Write the READ_SINGLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

10. To meet timing restrictions related to block read operations, it is advisable to not wait for the CMD_RSP_END indication in the SDH_STATUS register. Instead move immediately on the next step. This is because the card can send data before a response has completed on the SD_CMD signal. Moving immediately to step 11 ensures that a receive FIFO overflow does not occur.

11. Poll the RX_FIFO_RDY bit or the RX_DAT_ZERO bit in the SDH_STATUS register. These indicate that the receive FIFO has data available or is empty. Continue to read data from the SDH_FIFO register until all 512 bytes have been read.

12. When all bytes have been read, wait for the DAT_BLK_END event. This indicates that the data was received correctly and passed the CRC check. The DAT_END event may also occur depending on the value written to SDH_DATA_LGTH.

13. Clear the DAT_BLK_END and DAT_END flags using the SDH_STATUS_CLR register.

Using DMA

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.

2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.

3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.

4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.

5. Configure the DMA channel assigned to the SDH controller. Write the DMAx_START_ADDR register with the address where the first byte of received data is to be stored. The DMAx_X_COUNT register should be set to 128 and the DMAx_X_MODIFY register to 4. The DMAx_CONFIG register should be set for DMA enable, a word size of 32-bits and direction set to memory write.

6. Write the number of bytes to be transferred to the SDH_DATA_LGTH register. This is 512 bytes for a single block.

7. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a read operation.

8. Write the source start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card and accesses are not enabled.

9. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E, DTX_DIR and DTX_DMA_E should also be set to enable the data path state machine, to set the direction of transfer from card to controller, and to allow the DMA controller access to the receive FIFO. All other fields of the SDH_DATA_CTL register should be zero.

10. Write the READ_SINGLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

11. Unlike with core accesses, it is safe to poll the CMD_RSP_END event and once detected, to clear the status bit using the SDH_STATUS_CLR register. The DMA controller, enabled in step 5, ensures that any data sent to the receive FIFO prior to the CMD_RSP_END event is received correctly.

12. Wait for the DAT_BLK_END event. This indicates that the data was received correctly and has passed the CRC check. The DAT_END event may also be set depending on the value written to SDH_DATA_LGTH.

13. Clear the DAT_BLK_END and DAT_END flags using the SDH_STATUS_CLR register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register if applicable.

Multiple Block Write Operations

Block write operations typically consist of 512 bytes of data per block. If the card supports other block lengths or the default block length as specified in the CID register is not 512—then the block length of the SDH must be configured accordingly. The block length of the card and the block length of the SDH must be configured for the same block size at all times. The block length of the SDH is configured using the DTX_BLK_LGTH field of the SDH_DATA_CTL register.

Using Core

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.

3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.
5. Write to the SDH_DATA_LGTH register, the number of bytes to be transferred. For example 4096—to write eight blocks of 512 bytes.
6. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a write operation.
7. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card and accesses are not enabled.
8. Write the WRITE_MULTIPLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.
9. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
10. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E should also be set to enable the data path state machine. All other fields of the SDH_DATA_CTL register should be zero.

11. Write data to the SDH_FIFO register until the FIFO is full as indicated by the TX_FIFO_FULL flag of the SDH_STATUS register. Continue to write data to the FIFO as long as the FIFO is not full or write data in blocks of eight 32-bit words if polling on the TX_FIFO_STAT bit indicates the transmit FIFO is half empty. Continue to write data until all 128 32-bit words (512 bytes) have been transferred.

12. Wait for the DAT_BLK_END event that indicates the card has responded with the CRC token.

13. Clear the DAT_BLK_END flag

14. Repeat steps 11 to 13 for the number of blocks to be transferred or until the DAT_END event occurs. When waiting for the DAT_END event to occur, move to the next step only when the following DAT_BLK_END event has occurred.

Pay particular attention to this step. The DAT_END event occurs when the SDH_DATA_CNT register decrements to zero. At this point, the SDH has emptied the FIFO but is waiting for the card to send the CRC token back for the block. It is only safe to send out the STOP_TRANSMISSION command when the DAT_BLK_END event that follows the DAT_END event has occurred. Failure to wait for both of these events may result in the SDH sending the STOP_TRANSMISSION command before receiving the CRC response. This would result in the card treating the final data block as incomplete and thus the final block would not be programmed.

15. Write the STOP_TRANSMISSION command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

16. Clear the DAT_END flag using the SDH_STATUS_CLR register.

Using DMA

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.
3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.
5. Configure the DMA channel assigned to the SDH controller. Write DMAx_START_ADDR with the address of the first byte of data to be written to the card. The DMAx_X_COUNT register should be set to the overall number of 32-bit words to be written, for this example 1024 for the transfer of 4096 bytes. The DMAx_X_MODIFY register should be set to 4. The DMAx_CONFIG register should be set for DMA enable and a word size of 32-bits.
6. Once the DMA channel has been configured and enabled, write to the SDH_DATA_LGTH register, the number of bytes to be transferred. For example 4096—to write eight blocks of 512 bytes.
7. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a write operation.
8. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card and accesses are not enabled.

9. Write the WRITE_MULTIPLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

10. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.

11. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E and DTX_DMA_E should also be set to enable the data path state machine, and to allow the DMA controller to access the transmit FIFO. All other fields of the SDH_DATA_CTL register should be zero.

12. Wait for the DAT_BLK_END events that will be set upon successful completion of each block transfer. For a 4096 byte transfer, DAT_BLK_END is set eight times and must be cleared after it is detected using the SDH_STATUS_CLR register. Either count the number of DAT_BLK_EVENTS that occurred and move to the next step once the expected count has been reached or keep processing DAT_BLK_END events until the DAT_END event occurs and move to the next step on the next DAT_BLK_END event following the DAT_END event.

Pay particular attention to this step. The DAT_END event occurs when the SDH_DATA_CNT register decrements to zero. At this point, the SDH has emptied the FIFO but is waiting for the card to send the CRC token back for the block. It is only safe to send out the STOP_TRANSMISSION command when the DAT_BLK_END event that follows the DAT_END event has occurred. Failure to wait for both of these events may result in the SDH sending the STOP_TRANSMISSION command before receiving the CRC response. This would result in the card treating the final data block as incomplete and thus the final block would not be programmed.

13. Write the STOP_TRANSMISSION command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.
14. Clear the DAT_END flag using the SDH_STATUS_CLR register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register if applicable.

Multiple Block Read Operations

Block read operations typically consist of 512 bytes of data per block. If the card supports other block lengths or the default block length as specified in the CID register is not 512 then the block length of the SDH must be configured accordingly. The block length of the card and the block length of the SDH must be configured for the same block size at all times. The block length of the SDH is configured using the DTX_BLK_LGTH field of the SDH_DATA_CTL register.

Using Core

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.
3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.

5. Write to the SDH_DATA_LGTH register, the number of bytes to be transferred. For example 4096—to write eight blocks of 512 bytes.
6. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a read operation.
7. Write the destination start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card and accesses are not enabled.
8. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E and DTX_DIR should also be set to enable the data path state machine, and to set the direction of transfer from card to controller. All other fields of the SDH_DATA_CTL register should be zero.
9. Write the READ_MULTIPLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.
10. To meet timing restrictions related to block read operations, it is advisable to not wait for the CMD_RSP_END indication in the SDH_STATUS register. Instead move immediately on the next step. This is because the card can send data before a response has completed on the SD_CMD signal. Moving immediately to step 11 ensures that a receive FIFO overflow does not occur.
11. Poll the RX_FIFO_RDY bit or the RX_DAT_ZERO bit in the SDH_STATUS register. These indicate that the receive FIFO has data available or is empty. Continue to read data from the SDH_FIFO register until all the bytes in the block have been read.

12. When all bytes have been read, wait for the DAT_BLK_END event. This indicates that the data was received correctly and passed the CRC check. Clear the DAT_BLK_END flag using SDH_STATUS_CLR register.
13. Repeat step 11 to step 13 until the required number of blocks have been read or until the DAT_END event has occurred.
14. Clear the DAT_END flag using the SDH_STATUS_CLR register.

Using DMA

1. Write the 16-bit RCA of the card to the upper 16-bits of the SDH_ARGUMENT register.
2. Write the SELECT/DESELECT_CARD command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1b.
3. Wait for the CMD_RSP_END indication in the SDH_STATUS register. When the indication is detected, clear the status bit using the SDH_STATUS_CLR register.
4. Verify the response in the SDH_RESPONSE0 register to ensure that the device is not busy and that no errors have occurred.
5. Configure the DMA channel assigned to the SDH controller. Write the DMAx_START_ADDR register with the address where the first byte of received data is to be stored. The DMAx_X_COUNT register should be set to the number of 32-bit words to be read. This would be 1024 for a 4096 byte read transfer. The DMAx_X_MODIFY register should be set to 4. The DMAx_CONFIG register should be set for DMA enable, a word size of 32-bits and direction set to memory write.

6. Write to the SDH_DATA_LGTH register, the number of bytes to be transferred. This will be 4096 for eight blocks of 512 bytes.
7. Write to the SDH_DATA_TIMER register, the appropriate timeout value for a read operation.
8. Write the source start address to the SDH_ARGUMENT register. The address must be aligned to a 512 byte boundary. If the address is misaligned, the card is not a high capacity SD card or sector addressable MMC card and accesses are not enabled.
9. Enable the data path state machine by setting the DTX_BLK_LGTH bits in the SDH_DATA_CTL register to 9 for a 512 byte block. DTX_E, DTX_DIR and DTX_DMA_E should also be set to enable the data path state machine, to set the direction of transfer from card to controller, and to allow the DMA controller access to the receive FIFO. All other fields of the SDH_DATA_CTL register should be zero.
10. Write the READ_MULTIPLE_BLOCK command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.
11. Unlike with core accesses, it is safe to poll the CMD_RSP_END event and once detected, to clear the status bit using the SDH_STATUS_CLR register. The DMA controller, enabled in step 5, ensures that any data sent to the receive FIFO prior to the CMD_RSP_END event is received correctly.
12. Wait for either the expected number of instances of the DAT_BLK_END event that will be set on successful completion of each block transfer—or the DAT_END event. For a 4096 byte transfer and a block size of 512 bytes, the DAT_BLK_END event occurs eight times and it should be cleared after it is detected using the SDH_STATUS_CLR register.

13. Write the STOP_TRANSMISSION command to the SDH_COMMAND register. This configures the command path state machine to expect a short response by setting CMD_RSP and clearing CMD_L_RSP. The response type is R1.

14. Clear the DAT_END flag using the SDH_STATUS_CLR register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register if applicable.

SDH Registers

The SDH memory-mapped registers (MMRs) regulate the operation of the SDH. Descriptions and bit diagrams for each of these MMRs are provided in the following section.

Table 27-10 lists the SDH memory-mapped registers starting at base address 0xFFC03900.

Table 27-10. SDH Functional Registers

Register Name	Address	Type	Access	Description
SDH_PWR_CTL	0xFFC0 3900	R/W	16-bit	"SDH Power Control Register (SDH_PWR_CTL)" on page 27-55
SDH_CLK_CTL	0xFFC0 3904	R/W	16-bit	"SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55
SDH_ARGUMENT	0xFFC0 3908	R/W	32-bit	"SDH Argument Register (SDH_ARGUMENT)" on page 27-57
SDH_COMMAND	0xFFC0 390C	R/W	16-bit	"SDH Command Register (SDH_COMMAND)" on page 27-57
SDH_RESP_CMD	0xFFC0 3910	R	16-bit	"SDH Response Command Register (SDH_RESP_CMD)" on page 27-58
SDH_RESPONSE0	0xFFC0 3904	R	32-bit	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
SDH_RESPONSE1	0xFFC0 3908	R	32-bit	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59

Table 27-10. SDH Functional Registers (Cont'd)

Register Name	Address	Type	Access	Description
SDH_RESPONSE2	0xFFC0 391C	R	32-bit	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
SDH_RESPONSE3	0xFFC0 3920	R	32-bit	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
SDH_DATA_TIMER	0xFFC0 3924	R/W	32-bit	"SDH Data Timer Register (SDH_DATA_TIMER)" on page 27-60
SDH_DATA_LGTH	0xFFC0 3928	R/W	16-bit	"SDH Data Length Register (SDH_DATA_LGTH)" on page 27-61
SDH_DATA_CTL	0xFFC0 392C	R/W	16-bit	"SDH Data Control Register (SDH_DATA_CTL)" on page 27-61
SDH_DATA_CNT	0xFFC0 3930	R	16-bit	"SDH Data Counter Register (SDH_DATA_CNT)" on page 27-62
SDH_STATUS	0xFFC0 3934	R	32-bit	"SDH Status Register (SDH_STATUS)" on page 27-63
SDH_STATUS_CLR	0xFFC0 3938	W1A	16-bit	"SDH Status Clear Register (SDH_STATUS_CLR)" on page 27-65
SDH_MASK0	0xFFC0 393C	R/W	32-bit	"SDH Interrupt Mask Registers (SDH_MASKx)" on page 27-66
SDH_MASK1	0xFFC0 3940	R/W	32-bit	"SDH Interrupt Mask Registers (SDH_MASKx)" on page 27-66
SDH_FIFO_CNT	0xFFC0 3948	R	16-bit	"SDH FIFO Counter Register (SDH_FIFO_CNT)" on page 27-68
SDH_FIFOx	0xFFC0 3980	R/W	32-bit	"SDH Data FIFO Register (SDH_FIFO)" on page 27-69
SDH_E_STATUS	0xFFC0 39C0	R/W1C	16-bit	"SDH Exception Status Register (SDH_E_STATUS)" on page 27-69
SDH_E_MASK	0xFFC0 39C4	R/W	16-bit	"SDH Exception Mask Register (SDH_E_MASK)" on page 27-70

Table 27-10. SDH Functional Registers (Cont'd)

Register Name	Address	Type	Access	Description
SDH_CFG	0xFFC0 39C8	R/W	16-bit	"SDH Configuration Register (SDH_CFG)" on page 27-71
SDH_RD_WAIT_EN	0xFFC0 39CC	R/W	16-bit	"SDH Read Wait Enable Register (SDH_RD_WAIT_EN)" on page 27-72
SDH_PID0	0xFFC0 39D0	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID1	0xFFC0 39D4	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID2	0xFFC0 39D8	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID3	0xFFC0 39DC	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID4	0xFFC0 39E0	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID5	0xFFC0 39E4	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID6	0xFFC0 39E8	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73
SDH_PID7	0xFFC0 39EC	R	16-bit	"SDH Identification Registers (SDH_PIDx)" on page 27-73

SDH Power Control Register (SDH_PWR_CTL)

The SDH_PWR_CTL register contains bits that control the power to the SDH module as well as the open-drain configuration for the SDH_CMD signal. The PWR_ON field must be set to b#11 to enable the SDH. The SDH_CMD_OD bit, when set, drives the SDH_CMD signal to open-drain mode. The default mode of operation is push-pull. After a data write, data cannot be written to this register for five SCLK cycles.

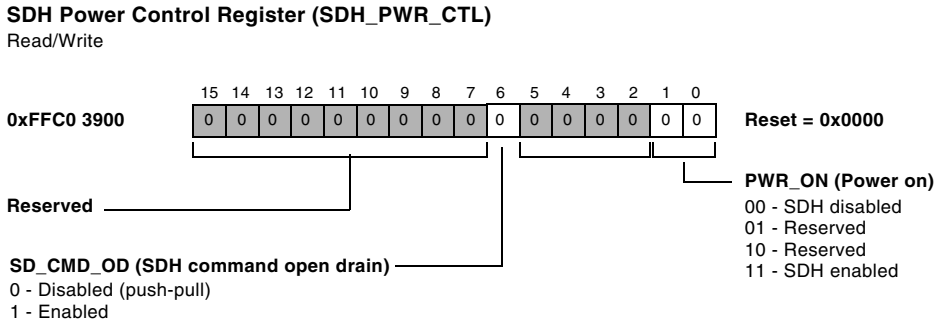


Figure 27-6. SDH Power Control Register

SDH Clock Control Register (SDH_CLK_CTL)

The SDH_CLK_CTL register controls the SDH clock. SDH_CLK is derived directly from the SCLK by enabling CLKDIV_BYPASS. Otherwise SDH_CLK is determined by the current SCLK frequency and the CLKDIV field as shown in Equation 27-1.

Equation 27-1. SDH Clock Frequency

$$SDH\_CLK = \frac{SCLK}{2 \times (CLKDIV + 1)}$$

To conserve power the SDH clock can be disabled without disabling the entire SDH interface by using the CLK_EN bit. Additionally, when the PWR_SV_EN bit is set, the SDH_CLK signal is only driven when the SDH is performing a transfer either to or from the card. The data bus width of the SDH interface is also controlled by this register.

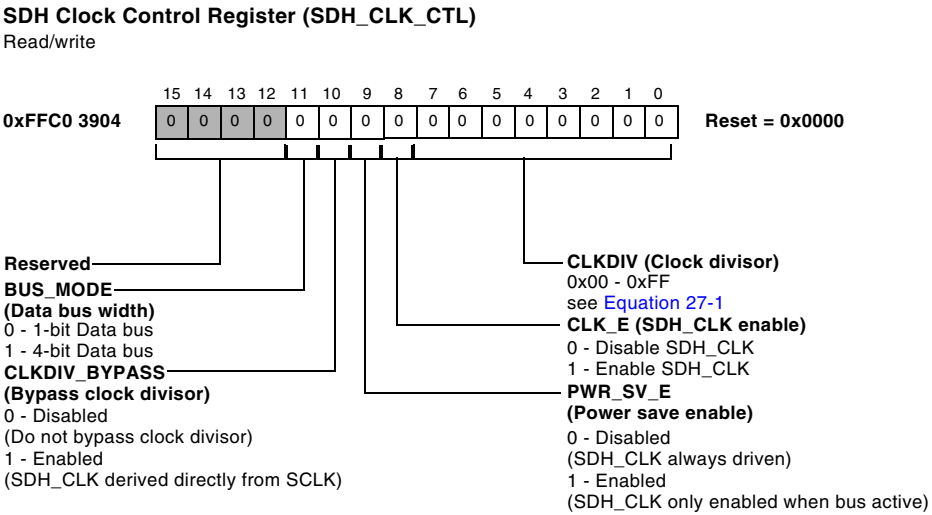


Figure 27-7. SDH Power Control Register

SDH Argument Register (SDH_ARGUMENT)

The SDH_ARGUMENT register contains the 32-bit argument that is sent on the SDH_CMD signal as part of a command message. If a command requires an argument, the argument must first be loaded into the SDH_ARGUMENT register prior to writing and enabling the command in the SDH_COMMAND register.

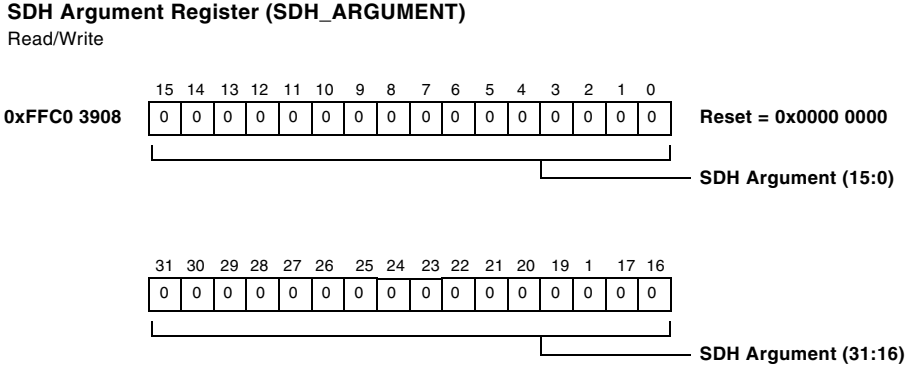


Figure 27-8. SDH Argument Register

SDH Command Register (SDH_COMMAND)

The SDH_COMMAND register controls the command path state machine. The CMD_IDX field contains the index of the command to be issued by the SDH as part of the command message. If the command requires a response, this is indicated by CMD_RSP.

The length of the response (short or long) is controlled by the CMD_L_RSP field. The command path state machine is active when the CMD_E bit is set and disabled if this bit is cleared.

The CMD_E bit does not have to be manually cleared after the command sequence has completed. The command path state machine automatically terminates and goes to IDLE when the operation has completed.

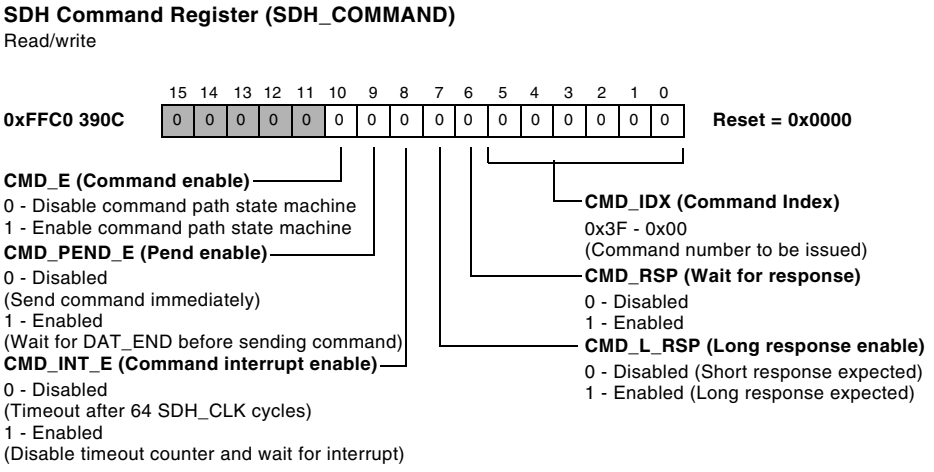


Figure 27-9. SDH Command Register

SDH Response Command Register (SDH_RESP_CMD)

The SDH_RESP_CMD register contains the command index field of the last response received. If the command response does not contain a command index field, as is the case with a long response, the RESP_CMD field would typically be ignored. In this situation it will likely contain b#111111 which is the value of the reserved field of the response.

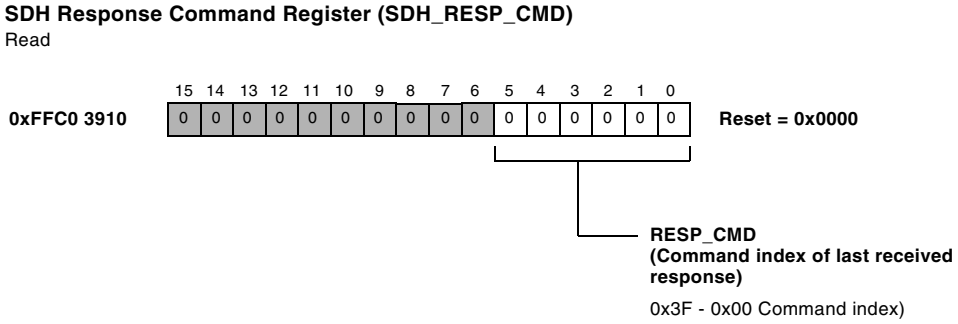


Figure 27-10. SDH Command Response Register

SDH Response Registers (SDH_RESPONSEx)

The SDH_RESPONSEx registers are four registers that contain the response information received back from a card for a given command message. The received response may be 32 or 127 bits in length depending on whether the response type is short or long. The most significant bit of the response is received first and is located in bit 31 of the SDH_RESPONSE0 register. Bit 0 of SDH_RESPONSE3 is always zero. Table 27-11 shows the SDH response registers contents for the two types of responses.

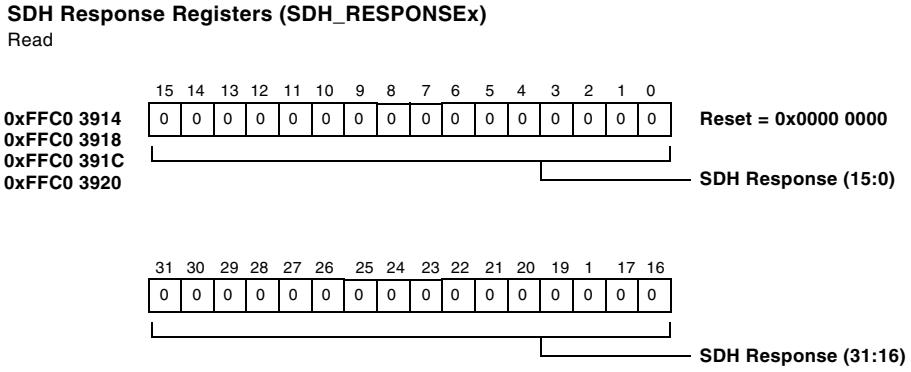


Figure 27-11. SDH Response Registers

Table 27-11. SDH Response Register Contents

Response Register	Short Response	Long Response
SDH_RESPONSE0	Response bits [31:0]	Response bits [127:96]
SDH_RESPONSE1	not used	Response bits [95:64]
SDH_RESPONSE2	not used	Response bits [63:32]
SDH_RESPONSE3	not used	Response bits [31:0]

SDH Data Timer Register (SDH_DATA_TIMER)

The SDH_DATA_TIMER register contains a 32-bit value for the data timeout period in SDH_CLK cycles. An internal counter loads the value of this register, and starts to decrement when the data path state machine enters the WAIT_R or the BUSY states. If the timer decrements to zero while the data path state machine is still in either of these two states, the DAT_TIMEOUT event is generated. The SDH_DATA_TIMER and the SDH_DATA_LGTH register must both be written prior to starting a data transfer with the SDH_DATA_CTL register.

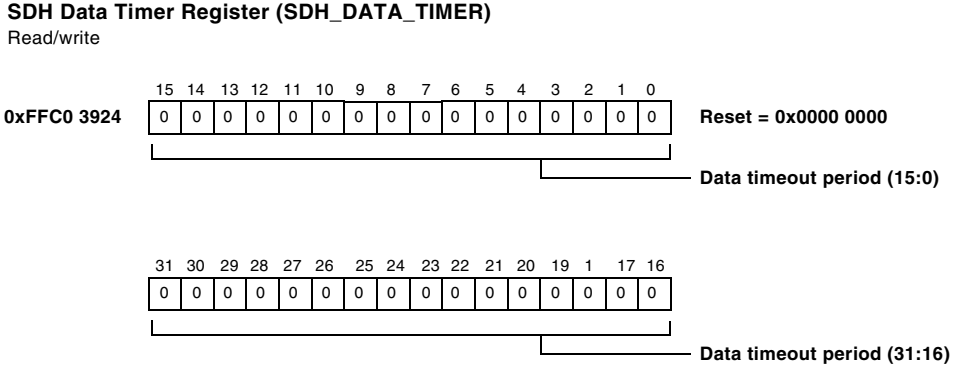


Figure 27-12. SDH Data Timer Register

SDH Data Length Register (SDH_DATA_LGTH)

The SDH_DATA_LGTH register contains a 16-bit value for the number of data bytes to be transferred before generating the DAT_END event. The value loaded to this register is copied into the SDH_DATA_CNT register when the data path state machine is enabled and starts the transfer.

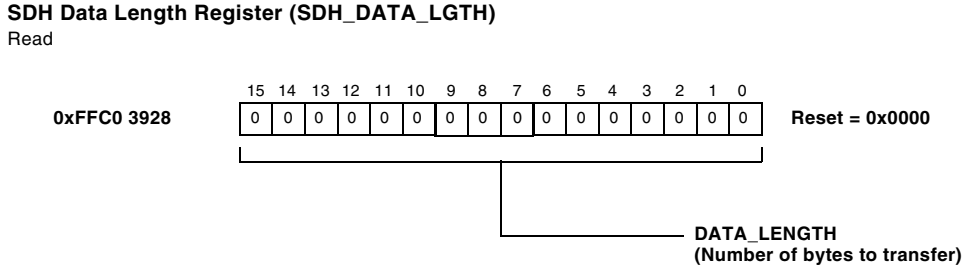


Figure 27-13. SDH Data Length Register

SDH Data Control Register (SDH_DATA_CTL)

The SDH_DATA_CTL register controls the data path state machine. The state machine becomes enabled once the DTX_E bit is set. The direction of the transfer is determined by DTX_DIR. If the DMA channel is to be used for the data transfer then the DTX_DMA_E bit must be set. Otherwise the SDH FIFO would only be accessible through the core. For block transfers, the block length must be specified by DTX_BLK_LGTH, where the block length is $2^{DATA\_BLK\_LGTH}$. After a data write, data cannot be written to this register for five SCLK cycles.

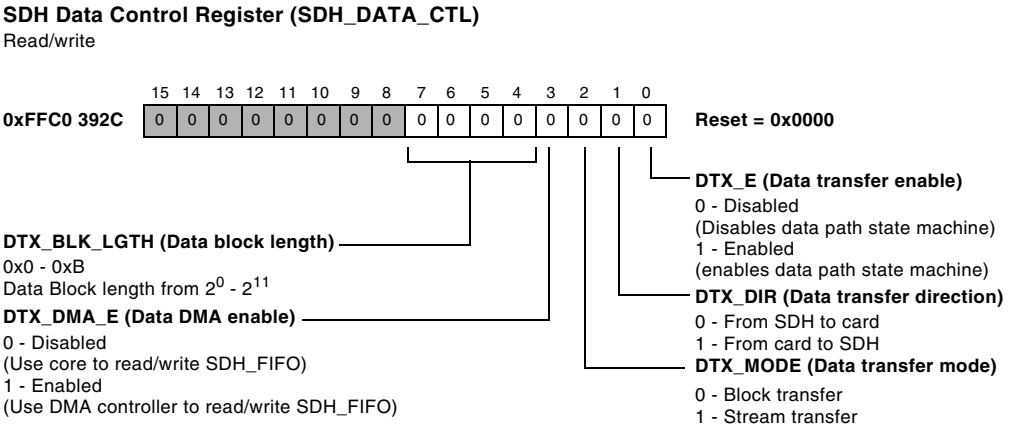


Figure 27-14. SDH Data Control Register

SDH Data Counter Register (SDH_DATA_CNT)

The SDH_DATA_CNT register gets loaded from the SDH_DATA_LGTH register when the data path state machine becomes enabled and moves from the IDLE state to the WAIT_S or WAIT_R states. As the data is transferred, the counter decrements. When it decrements to zero, the state machine moves back to the IDLE state and the DAT_END event occurs.

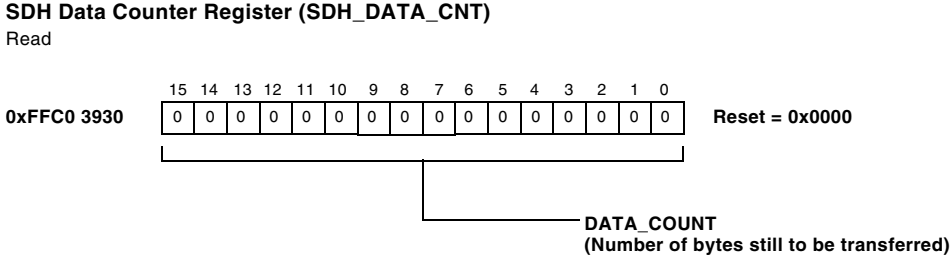


Figure 27-15. SDH Data Counter Register

SDH Status Register (SDH_STATUS)

The SDH_STATUS register contains both static and dynamic flags that indicate the status of the SDH. The static flags (bits[10:0]) remain asserted and must be cleared by writing to the SDH_STATUS_CLR register. The dynamic flags (bits[21:11]) change state depending on the state of the underlying logic. Transmit and receive FIFO logic control bits[21:12] vary depending on the state of the FIFO and whether the FIFO is currently enabled for a transmit or receive operation.

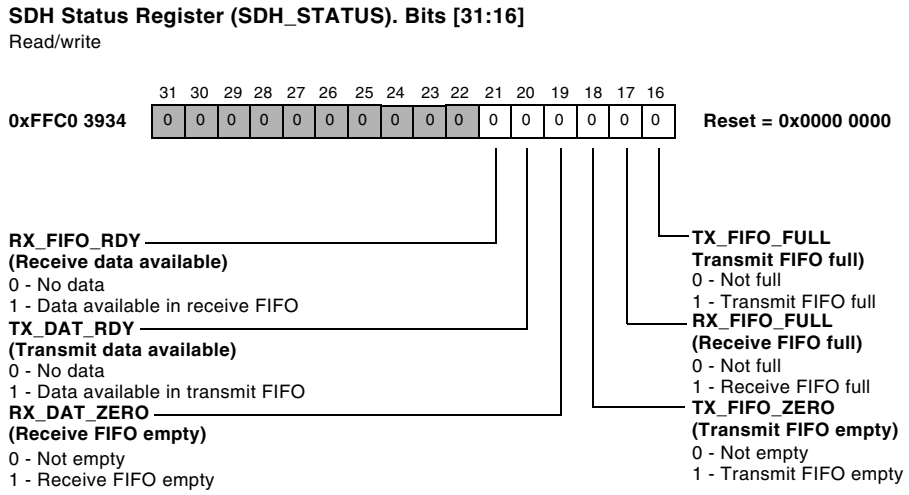


Figure 27-16. SDH Status Register, Bits [31:16]

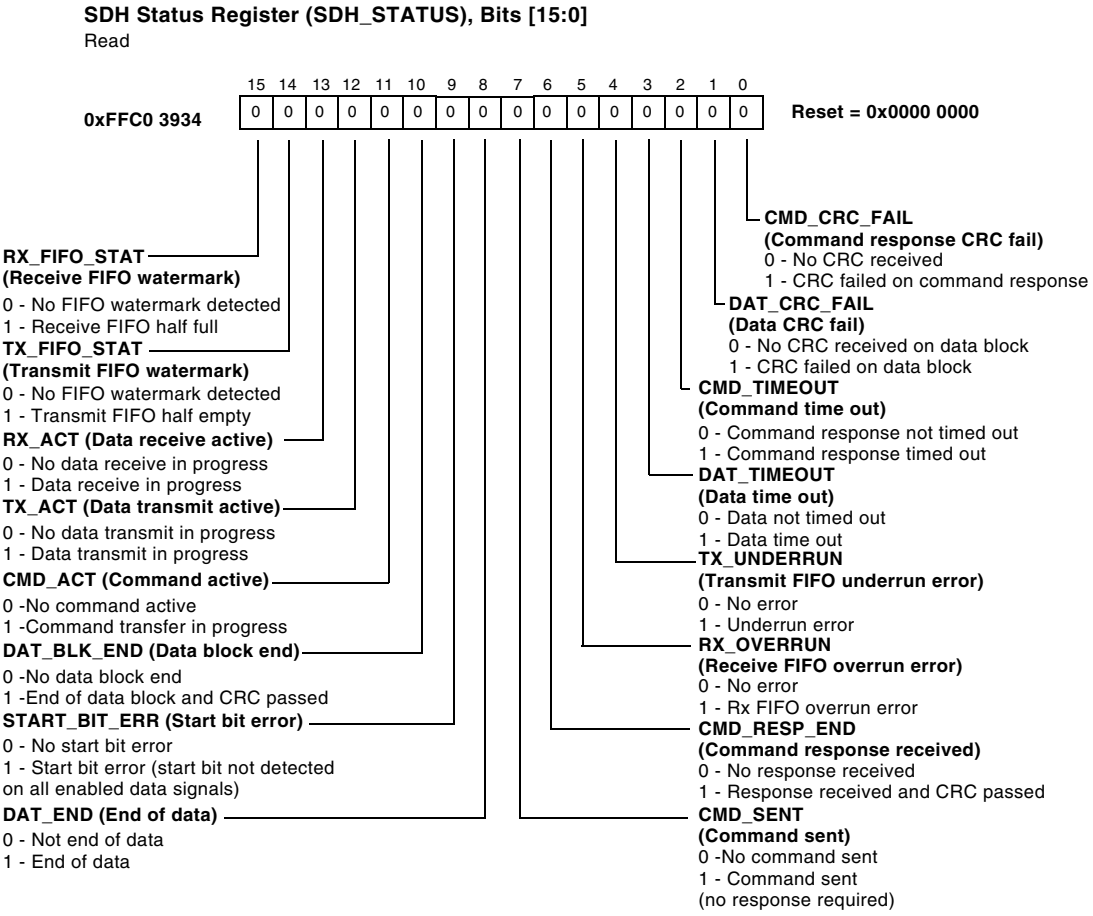


Figure 27-17. SDH Status Register, Bits[15:0]

SDH Status Clear Register (SDH_STATUS_CLR)

The SDH_STATUS_CLR register is used to clear the static flags of the SDH_STATUS register. Writing a b#1 to any of the bits, clears the corresponding SDH_STATUS flag.

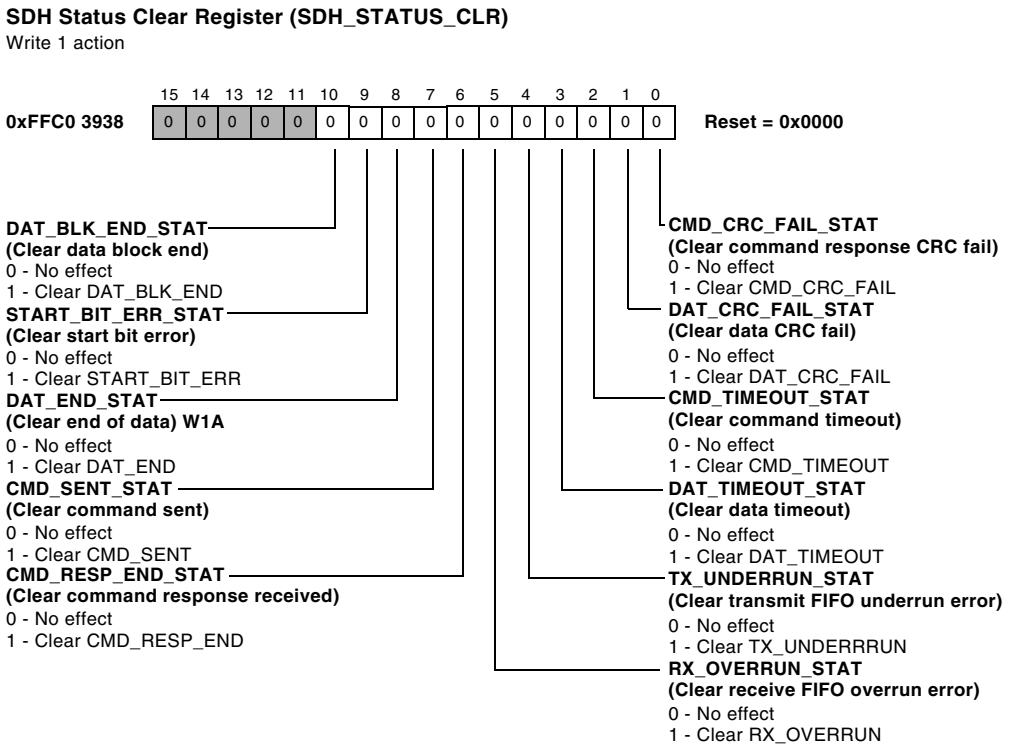


Figure 27-18. SDH Status Clear Register

SDH Interrupt Mask Registers (SDH_MASKx)

The SDH_MASKx registers determine which of the static and dynamic flags of the SDH_STATUS register generate a request to the SIC for one of the two available SDH interrupts. An interrupt is enabled by setting the corresponding bit in the SDH_MASKx register to 1. Interrupts enabled by the SDH_MASK0 register send an IRQ0 interrupt, and interrupts enabled by the SDH_MASK1 register send an IRQ1 interrupt.

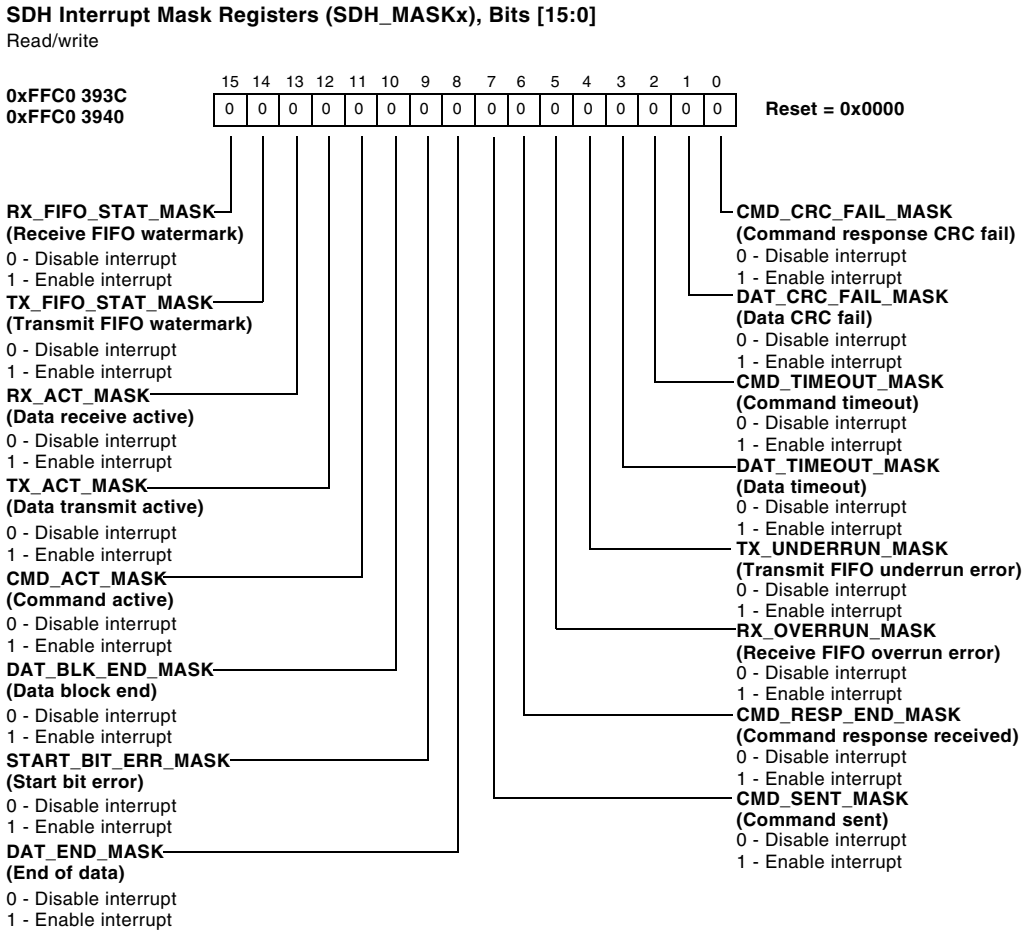


Figure 27-19. SDH Interrupt Mask Registers, Bits [15:0]

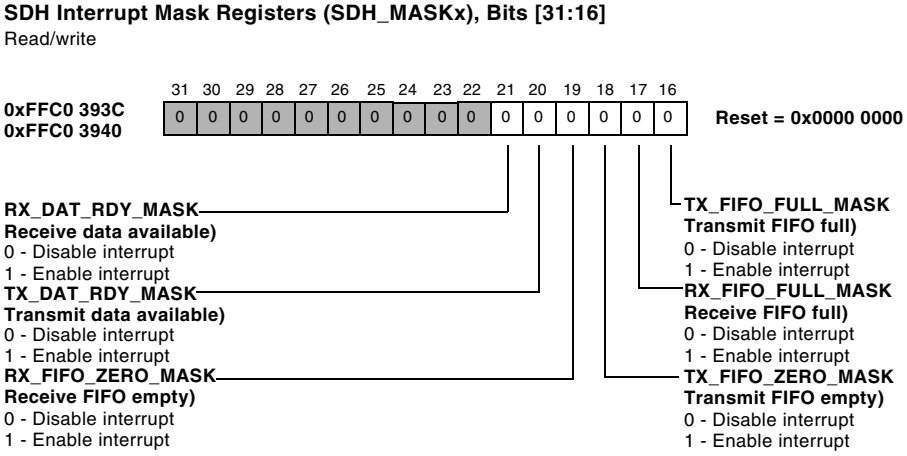


Figure 27-20. SDH Interrupt Mask Registers, Bits [31:16]

SDH FIFO Counter Register (SDH_FIFO_CNT)

The SDH_FIFO_CNT register contains a value indicating the number of words still to be read from or written to the FIFO. The SDH_FIFO_CNT register is loaded from the SDH_DATA_LGTH register when the DTX_E bit of the SDH_DATA_CONTROL register is set. If the data length is not word aligned (multiple of 4) the remaining 1 to 3 bytes are regarded as a word.

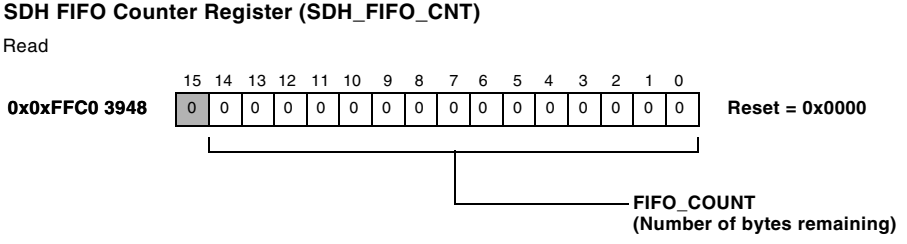


Figure 27-21. SDH FIFO Counter Register

SDH Data FIFO Register (SDH_FIFO)

The SDH_FIFO register provides access to the 16 entry transmit and receive FIFO. The register is accessed as a 32-bit word.

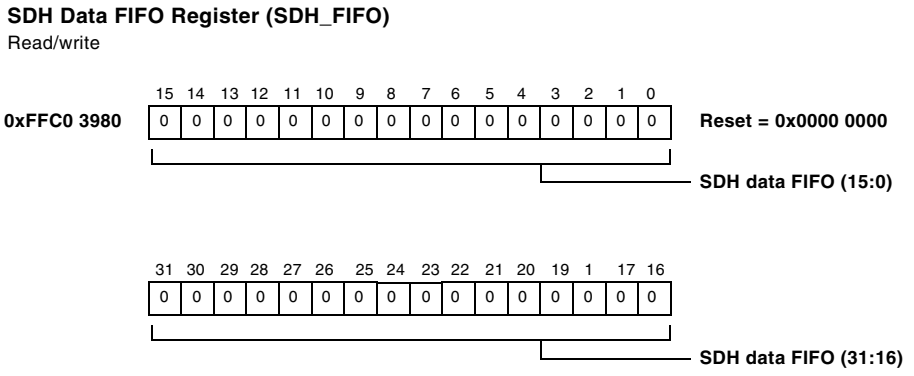


Figure 27-22. SDH Data FIFO Register

SDH Exception Status Register (SDH_E_STATUS)

The SDH_E_STATUS register contains exception status bits for SD/SDIO cards and the card detection logic. These status bits can be used to generate an interrupt request using the IRQ0 signal by enabling the interrupt in the SDH_E_MASK register. All bits in this register are write-1-clear bits. The SDIO_INT_DET interrupt is an interrupt generated by SDIO cards on the SDH_DATA1 signal. The SD_CARD_DET bit is set when a rising edge is detected on the SDH_DATA3 signal and is intended for use with MMC devices that support card detection using this signal.

Most SD/MMC sockets contain two additional signals for card detect and write protect functionality. It is highly recommended that card detection be implemented by using these signals. Figure 27-2 on page 27-12 shows a typical interface between the SDH interface and the card socket. The card detect signal should

be debounced and interfaced to a GPIO signal. This provides the most robust and reliable method of card detection and is compatible with all SD/SDIO and MMC devices. In addition to providing card detect functionality, it also allows for interrupt driven card removal detection.

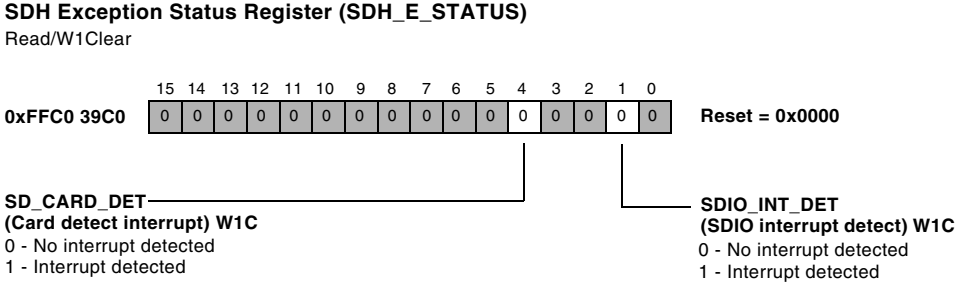


Figure 27-23. SDH Exception Status Register

SDH Exception Mask Register (SDH_E_MASK)

The SDH_E_MASK register contains mask bits for the SDH_E_STAT status bits. Writing a '1' to the SDH_E_MASK bit enables the interrupt for the corresponding bit in the SDH_E_STAT register.

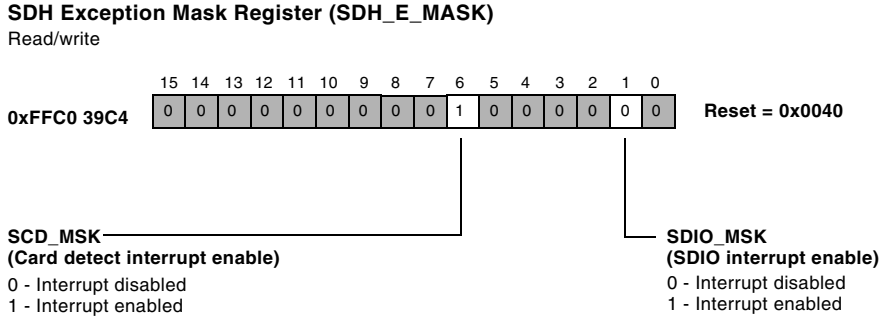


Figure 27-24. SDH Exception Mask Register

SDH Configuration Register (SDH_CFG)

The SDH_CFG register enables and disables portions of the SDH module. The CLKS_EN bit must be set in order to enable the SDH for operation.

After reset PD_SDDAT3 is set to enable pull-down of the SDH_DAT3 signal. This implements card detection if that feature is available. Once a card is detected, PD_SDDAT3 should be cleared and PUP_SDDAT3 should be enabled.

The pull-up and pull-down resistors on the SDH_DATAx signals only become active when the corresponding GPIO pins are configured for SDH functionality through pin multiplexing. For example, if only the 4-bit data bus is enabled in the pin multiplexing, then setting PUP_SDDAT only enables the pull-up resistors on the signals that are configured for SDH use.

The SDH_CFG register provides additional SDIO functionality.

Set SD4E to enable the SDIO 4-bit mode; in addition to setting the bus width to 4-bit using the WIDE_BUS bit of the SDH_CLK_CTL register.

Setting the MWE bit allows for SDIO interrupts to be detected outside the specified once cycle window. This should be set when interrupt support is required during multiple block read transactions from SDIO.

The SDH can also be reset with the SD_RST bit. Writing this bit resets the SDH module and sets all registers back to their default values.

PD_SDDAT3 and PUP_SDDAT3 are mutually exclusive and should never both be set at any given time. Always clear one before setting the other.

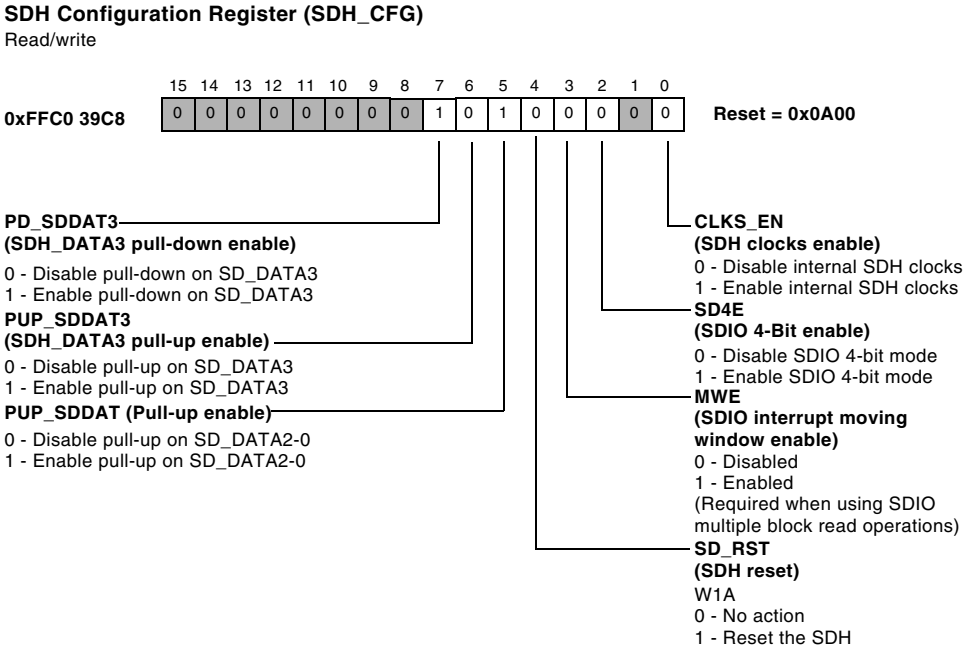


Figure 27-25. SDH Configuration Register

SDH Read Wait Enable Register (SDH_RD_WAIT_EN)

When the RWR bit in the SDH_RD_WAIT_EN register is set, a read-wait request is sent to the SDIO card. This bit must be cleared when the software is ready to resume the data transfer. The functionality applies to both 1-bit and 4-bit SDIO modes.

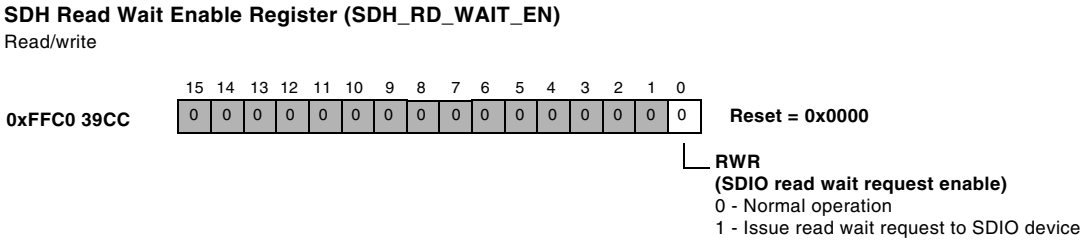


Figure 27-26. SDH Read Wait Enable Register

SDH Identification Registers (SDH_PIDx)

The SDH_PIDx registers contain a fixed value at reset and are used to identify the peripheral revision. There are a total of eight 16-bit identification registers of which the lower 8-bits are valid. The contents of these eight registers are listed in Table 27-12.

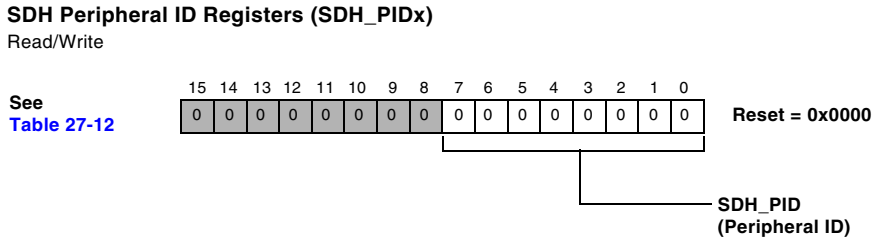


Figure 27-27. SDH Peripheral ID Registers

Table 27-12. SDH Peripheral ID

Address	SDH Peripheral ID Register	SDH_PID Value
0xFFC0 39D0	SDH_PID0	0x80
0xFFC0 39D4	SDH_PID1	0x11
0xFFC0 39D8	SDH_PID2	0x04
0xFFC0 39DC	SDH_PID3	0x00
0xFFC0 39E0	SDH_PID4	0x0D
0xFFC0 39E4	SDH_PID5	0xF0
0xFFC0 39E8	SDH_PID6	0x05
0xFFC0 39EC	SDH_PID7	0xB1

Programming Examples

Listing 27-1 shows an example initialization sequence to enable the use of the SDH.

Listing 27-1. SDH Port Register Configuration and SDH Initialization

```
_sdh_enable:

    /* save return address and frame pointer only */
    LINK 0;

    /*********************************************/
    /* Setup Base MMR address                    */
    /*********************************************/
    P1.L = 0x0000;
    P1.H = HI(PORTC_FER);

    /*********************************************/
```

```
    /* Disable the card detect and SDIO Interrupt as card detect
is implemented elsewhere via regular GPIO */
    /*********************************************/
    R3.L = 0;
    W[P1 + LO(SDH_E_MASK)] = R3;

    /*********************************************/
    /* Port Muxing for SDH                       */
    /*********************************************/
    R3.L = lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));
    R3.H = hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));
    [P1+LO(PORTC_MUX)] = R3;

    /*********************************************/
    /* Enable SDH functionality                  */
    /*********************************************/
    R3.L = PC13 | PC12 |PC11 | PC10 | PC9 | PC8;
    W[P1+LO(PORTC_FER)] = R3;
    ssync;

    /*********************************************/
    /* Clear card detect and SDIO status         */
    /*********************************************/
    R3 = W[P1 + LO(SDH_E_STATUS)];
    W[P1 + LO(SDH_E_STATUS)]= R3;

    /*********************************************/
    /* Clear any set status flags                */
    /*********************************************/
    R3 = W[P1 + LO(SDH_STATUS)];
    W[P1 + LO(SDH_STATUS_CLR)]= R3;

    /*********************************************/
    /* Enable SDH Clocks and configure resistors */
```

```
    /*********************************************/
    R3 = CLKS_EN | PUP_SDDAT | PUP_SDDAT3;
    W[P1 + LO(SDH_CFG)] = R3;

    /*********************************************/
    /* Turn on the SDH                           */
    /*********************************************/
    R3 = PWR_ON;
    W[P1+ LO(SDH_PWR_CTL)] = R3;

    /*********************************************/
    /* Enable the external clock and configure for power save
mode. Clock divisor = 0xA6 for <400KHz clock     */
    /*********************************************/
    R3 = PWR_SV_E | CLK_E | 0xA6;
    W[P1 + LO(SDH_CLK_CTL)] = R3;

    UNLINK;
    RTS;

_sdh_enable.END:
```

Listing 27-2 shows an example routine for sending a command to a device. The function is called with two passed parameters and returns 0 for a pass condition. If there is a failure, the function returns the status flag which is read from the SDH_STATUS register. Once the command is issued the routine polls on the SDH_STATUS register for either a pass or failure condition. Commands that require a response, return a CMD_CRC_FAIL or a CMD_TIMEOUT condition in the event of a failure. For a pass condition, meaning the command was received and the CRC passed, a CMD_RESP_END response is returned.

For commands that do not require a response, there are no failure conditions. Since the card provides no response at all, a pass condition is indicated by the CMD_SENT event.

Listing 27-2. SDH Send Command Example

```
/*********************************************************/
/*send_command(uint32_t Command, uint32_t Argument)*/
/*     */
/* Return: uint32_t error, 0 for success else the error */
/*                                                       */
/*Command format:
Bits 5:0 of the command are the command index to be sent.
Bit 6 of the command indicates if a response is expected
0 = No response
1 = Response
Bit 7 of the command indicates response type
0 = Short response
1 = Long Response                     */
/*                                                  *//* Descrip-
tion:                                                          */
/* Sends a command to the device. If the command expects a
response the error conditions are set to CRC Fail and Command
Timeout. The success condition is set to Command Response End
indicating the command was received. If the command does not
expect a response, there are no error conditions and the success
condition is set to Command Sent. The function returns 0 if there
was no error or it returns the error condition as per the
SDH_STATUS register. Only the success and error conditions are
then cleared in the status register thus maintaining any indica-
tions of other SDH status operations. This example could be
further enhanced to include a check on the received command index
from the response if one is expected and received.   */
/*                                                       */
/*********************************************************/
_send_command:
    LINK 0;
    [--SP] = (R7:4, P5:3);
```

```
    /*************************/
    /* R0 holds the Command */
    /* R1 holds the Argument */
    /*************************/

    /***************************************/
    /* Set base MMR for SDH register access */
    /***************************************/
    P5.L = 0;
    P5.H = HI(SDH_CLK_CTL);
   /*                */
    /*******************************************************/
    /* Configure error and success factors for the function

Default Errors are CMD_CRC_FAIL and CMD_TIMEOUT
Default Success conditions are CMD_RESP_END
If no response is expected from the sent command there are no
error conditions and the pass condition is CMD_SENT */
    /*******************************************************/
    _send_command_setup_success_error:
    /*******************************************************/
    /* Error and success factors if a response is expected */
    /*******************************************************/
    R5 = CMD_CRC_FAIL | CMD_TIMEOUT; /* Error */
    R4 = CMD_RESP_END;               /* Success */

    CC = BITTST(R0, BITPOS(CMD_RSP));

    IF CC JUMP _send_command_setup_success_error.end;

 /*********************************************************/
    /* Error and success factors if a response is not expected */
    /*********************************************************/
```

```
    R5 = 0;           /* Error */
    R4 = CMD_SENT; /* Success */
_send_command_setup_success_error.end:

R3 = R4 | R5; /* Combined error & success */

    /*******************************************************/
    /* Set the CMD_EN bit of the input command to be sent */
    /*******************************************************/
   bitset(R0,bitpos(CMD_E));

    /*************************************/
    /* Write the argument for the command */
    /*************************************/
   [P5 + LO(SDH_ARGUMENT)] = R1;

    /*************************************/
    /* Issue the command                 */
    /*************************************/
    W[P5 + LO(SDH_COMMAND)] = R0;

    /*******************************************************/
 /*Poll the SDH_STATUS register until a pass or fail condition */
    /*******************************************************/
_send_command_poll_loop:
    R7 = [P5 + LO(SDH_STATUS)];
    R6 = R7 & R3;
    CC = R6 == 0;
    IF CC JUMP _send_command_poll_loop;


/*************************************************************/
```

```
    /* Set default return value to success, then check for an
error condition. If it is an error condition set the return
parameter to the error condition                    */
    /************************************************/
   R0 = 0;
   R7 = R6 & R5;
   CC = R7 == 0;
   IF CC JUMP _send_command_clear_status;
   R0 = R6 & R5;

    /*****************************************/
    /* Clear the pass or fail flags and preserve all other status
flags   */
    /*****************************************/
    _send_command_clear_status:
    W[P5 + LO(SDH_STATUS_CLR)] = R3;

    (R7:4, P5:3) = [SP++];
    UNLINK;
    RTS;
_send_command.end:
```

Listing 27-3 shows a very basic routine for SD card identification. Note that the example is simplified and is not compliant with version 2.0 of the SD specification. It is included for illustrative purposes only.

Listing 27-3. SD Card Identification Example

```
_sd_identification:
    LINK 0;
    [--SP] = (R7:4, P5:3);
```

```
    /*********************************/
    /* Send the IDLE command         */
    /*********************************/
    R0 = SD_MMC_CMD_GO_IDLE_STATE;
    R1 = 0;
    CALL _send_command;

/*********************************/
/* Fetch card OCR register       */
/*********************************/
 _sd_identification_request_ocr:

    /***************/
    /* Send CMD55 */
    /***************/
    R0 = SD_MMC_CMD_APP_CMD;
    R1.L = LO(0x00000000);
    R1.H = HI(0x00000000);
    CALL _send_command;

    CC = R0 == 0;
    IF !CC JUMP _error;

    /***************/
    /* Send ACMD41 */
    /***************/
    R0 = SD_CMD_GET_OCR_VALUE;
    R1.L = LO(0x00FF8000);
    R1.H = HI(0x00FF8000);
    CALL _send_command;

    /*************************************/
```

```
    /* An R3 response has no valid CRC field so although a
response is required this R3 response will always generate a
CMD_CRC_FAIL condition. So this is special pass condition */
    /****************************************/
    CC = R0 == CMD_CRC_FAIL;
    IF !CC JUMP _error;

    P5.L = 0;
    P5.H = HI(SDH_RESPONSE0);

    /****************************************/
    /* Keep requesting the OCR register until the OCR register
indicates card is no longer busy                    */
    /****************************************/
    R7 = [P5 + LO(SDH_RESPONSE0)];
    CC = BITTST(R7, 31);
    IF !CC JUMP _sd_identification_request_ocr;


_sd_identification_request_cid:
    /***************/
    /* Send CMD2   */
    /***************/
    R0 = SD_MMC_CMD_ALL_SEND_CID;
    R1 = 0(X);
    CALL _send_command;
    CC = R0 == 0;
    IF !CC JUMP _error;

_sd_identification_request_rca:
    /***************/
    /* Send CMD3   */
    /***************/
    R0 = SD_CMD_SEND_RELATIVE_ADDR;
```

```
    R1 = 0(X);
    CALL _send_command;
    CC = R0 == 0;
    IF !CC JUMP _error;

    R7 = [P5 + LO(SDH_RESPONSE0)];

    R6.L = LO(0xFFFF0000);
    R6.H = HI(0xFFFF0000);

    R7 = R6 & R7;

    /***************************************/
    /* Save the RCA to a 32-bit parameter. The upper 16-bits con-
tain the RCA. We masked out the lower bits allowing for easy
logic OR operations for commands that require the RCA in their
argument field.   */
    /***************************************/
    P5.L = LO(_rca);
    P5.H = HI(_rca);
    [P5] = R7;

    (R7:4, P5:3) = [SP++];

    UNLINK;
    /* Return from subroutine */
    RTS;
 _sd_identification.END:
```

Listing 27-4 shows a very basic block write routine of 512 bytes through the use of the SDH DMA channel. This is a simplified example intended for illustrative purposes only.

Listing 27-4. Single Block Write using DMA

```
_sdh_single_block_write:
    LINK 0;
    [--SP] = (R7:4, P5:3);

    P4.L = LO(_rca);
    P4.H = HI(_rca);
    R1 = [P4];

    /***************/
    /* Send CMD7   */
    /***************/
    R0 = SD_MMC_CMD_SELECT_DESELECT_CARD;
    CALL _send_command;
    CC = R0 == 0;
    IF !CC JUMP _error;

    P5.L = 0;
    P5.H = HI(SDH_PWR_CTL);

    /************************************************/
    /* R0 contains the block to write to on the card so convert
this to a byte address */
    /************************************************/
    R6 = R0 << 9;   /* Convert the block to a byte address */
    R0 = 0(X);      /* Write transaction */
    R2 = 512(Z);    /* Default block size is 512 */

    /**********************************/
    /* Set the data length to 512 bytes */
    /**********************************/
    W[P5 + LO(SDH_DATA_LGTH)] = R2;
```

```
    /************************************************/
    /* Set up the timeout value. See SD Spec for details on how to
calculate the correct value.          */
    /************************************************/
    R7.H = HI(0x00FFFFFF);
    R7.L = LO(0x00FFFFFF);
    [P5 + LO(SDH_DATA_TIMER)] = R7;

    R1.L = LO(_write_buffer);
    R1.H = HI(_write_buffer);

    /*********************/
    /* Configure the DMA */
    /*********************/
    R7 = 0(X);
    W[P5 + LO(DMA22_CONFIG)] = R7;

    [P5 + LO(DMA22_START_ADDR)] = R1;

    R2 >>= 2;

    W[P5 + LO(DMA22_X_COUNT)] = R2;

    R7 = 4(X);
    W[P5 + LO(DMA22_X_MODIFY)] = R7;

    R7 = DMAEN | WDSIZE_32 |DI_EN;
    W[P5 + LO(DMA22_CONFIG)] = R7;

    /***********************************/
    /* Issue the write block command    */
    /***********************************/
    R0 = SD_MMC_CMD_WRITE_SINGLE_BLOCK;
    R1 = R6;
```

```
CALL _send_command;
CC = R0 == 0;
IF !CC JUMP _error;

/***********************************/
/* Start the transfer              */
/***********************************/
R7.L = LO(0x0099);
W[P5 + LO(SDH_DATA_CTL)] = R7;


R6.L = LO(DAT_END | DAT_BLK_END);
R6.H = HI(DAT_END | DAT_BLK_END);


/***********************************/
/* Wait for block completion       */
/***********************************/
_sdh_single_block_write_wait_data_end:
R7 = [P5 + LO(SDH_STATUS)];
R7 = R7 & R6;
CC = R7 == 0;
IF CC JUMP _sdh_single_block_write_wait_data_end;

/***********************************/
/* Clear SDH status                */
/***********************************/
W[P5 + LO(SDH_STATUS_CLR)] = R7;

/***********************************/
/* Clear DMA interrupt             */
/***********************************/
R7.L = LO(DMA_DONE);
W[P5 + LO(DMA22_IRQ_STATUS)] = R7;
```

```
    /*******************************************************/
    /* Keeps reading card status register until no longer busy */
    /*******************************************************/
    CALL _sd_wait_for_card_ready;

    (R7:4, P5:3) = [SP++];
    UNLINK;
    RTS;
_sdh_single_block_write.end:
```

Listing 27-5 shows a very basic block read routine of 512 bytes through the use of the SDH DMA channel. This is a simplified example intended for illustrative purposes only.

Listing 27-5. Single Block Read using DMA

```
_sdh_single_block_read:
    LINK 0;
    [--SP] = (R7:4, P5:3);

    P4.L = LO(_rca);
    P4.H = HI(_rca);
    R1 = [P4];

    /***************/
    /* Send CMD7   */
    /***************/
    R0 = SD_MMC_CMD_SELECT_DESELECT_CARD;
    CALL _send_command;
    CC = R0 == 0;
    IF !CC JUMP _error;

    P5.L = 0;
```

```
    P5.H = HI(SDH_PWR_CTL);

    /***********************************************/
    /* R0 contains the block to write to on the card so convert
this to a byte address.             */
    /***********************************************/
    R6 = R0 << 9;   /* Convert the block to a byte address */
    R0 = 0(X);      /* Write transaction */
    R2 = 512(Z);    /* Default block size is 512 */


    /*************************************/
    /* Set the data length to 512 bytes */
    /*************************************/
    W[P5 + LO(SDH_DATA_LGTH)] = R2;


    /***************************************************/
   /* Set up the timeout value. See SD Spec for details on how to
calculate the correct value.           */
    /***************************************************/
    R7.H = HI(0x00FFFFFF);
    R7.L = LO(0x00FFFFFF);
    [P5 + LO(SDH_DATA_TIMER)] = R7;

    R1.L = LO(_read_buffer);
    R1.H = HI(_read_buffer);

    /**********************/
    /* Configure the DMA */
    /**********************/
    R7 = 0(X);
    W[P5 + LO(DMA22_CONFIG)] = R7;
```

```
[P5 + LO(DMA22_START_ADDR)] = R1;

R2 >>= 2;

W[P5 + LO(DMA22_X_COUNT)] = R2;

R7 = 4(X);
W[P5 + LO(DMA22_X_MODIFY)] = R7;

R7 = DMAEN | WDSIZE_32 |DI_EN | WNR;
W[P5 + LO(DMA22_CONFIG)] = R7;

/************************************/
/* Issue the write block command    */
/************************************/
R0 = SD_MMC_CMD_READ_SINGLE_BLOCK;
R1 = R6;
CALL _send_command;
CC = R0 == 0;
IF !CC JUMP _error;

/************************************/
/* Start the transfer               */
/************************************/
R7.L = LO(0x009B);
W[P5 + LO(SDH_DATA_CTL)] = R7;


R6.L = LO(DAT_END | DAT_BLK_END);
R6.H = HI(DAT_END | DAT_BLK_END);


/************************************/
/* Wait for block completion        */
```

```
    /***********************************/
    _sdh_single_block_read_wait_data_end:
    R7 = [P5 + LO(SDH_STATUS)];
    R7 = R7 & R6;
    CC = R7 == 0;
    IF CC JUMP _sdh_single_block_read_wait_data_end;

    /***********************************/
    /* Clear SDH status                */
    /***********************************/
    W[P5 + LO(SDH_STATUS_CLR)] = R7;

    /***********************************/
    /* Clear DMA interrupt             */
    /***********************************/
    R7.L = LO(DMA_DONE);
    W[P5 + LO(DMA22_IRQ_STATUS)] = R7;


    /*****************************************************/
    /* Keeps reading card status register until no longer busy */
    /*****************************************************/
    CALL _sd_wait_for_card_ready;

    (R7:4, P5:3) = [SP++];
    UNLINK;
    RTS;
_sdh_single_block_read.end:
```

28 PIXEL COMPOSITOR

This chapter describes the pixel compositor (PIXC) and includes the following sections:

- "Overview" on page 28-1
- "Interface Overview" on page 28-2
- "Description of Operation" on page 28-4
- "Functional Description" on page 28-9
- "Programming Model" on page 28-34
- "PIXC Registers" on page 28-35

Overview

The pixel compositor (PIXC) for the ADSP-BF54x processor processor provides data overlay, transparent color, and color space conversion support for active (TFT) flat-panel digital color/monochrome LCD displays or analog NTSC/PAL video output. The color space conversion and text/graphic overlay capabilities, along with visual effect controls, such as transparency control, shorten the processing time on an image data stream, reduce power consumption and save system board space by removing the need for external glue logic.

The PIXC DMA channels should be configured for 32-bit transfers in order to ensure correct operation.

Features

The PIXC includes these features:

- Hardware-based graphics and text overlays
- YUV 4:2:2 or RGB888 input data formats
- Programmable color space conversion on the main image or the overlay image data path
- Overlay content transparency ratio control
- Transparent color, specified in the desired color space (RGB or YUV)
- Two DMA input channels and one DMA output channel
- Image data stream outputs for active-matrix TFT LCD panels or analog NTSC/PAL displays

Interface Overview

A top-level micro architecture diagram of the PIXC appears in Figure 28-1. As shown in Figure 28-1, the PIXC requires three DMA channels: one for the image data, one for the overlay data and one for storing the results back to L3, L2 or L1 memory. Frame C can also be fed back to the PIXC for multiple stages of processing, taking the place of frame A when this happens. The EPPI can then format the data for an LCD (for example, RGB888 to RGB666 or RGB565).

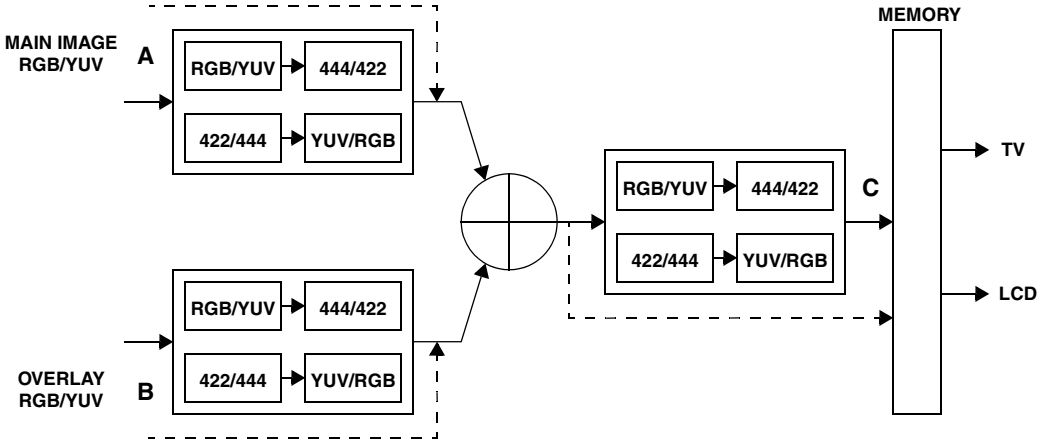


Figure 28-1. Pixel Compositor Top-Level Diagram

Only one color conversion or chroma resampling takes place for each given use of the PIXC block. The user can assume that a color space converter is present in each of the three places shown above. In reality, any input format (for both image and overlay) and any output format can be supported using a single appropriately-positioned color space converter.

Figure 28-2 shows a more detailed functional view of the PIXC block and shows the relevant connections to the DAB and system interrupt controller (SIC).

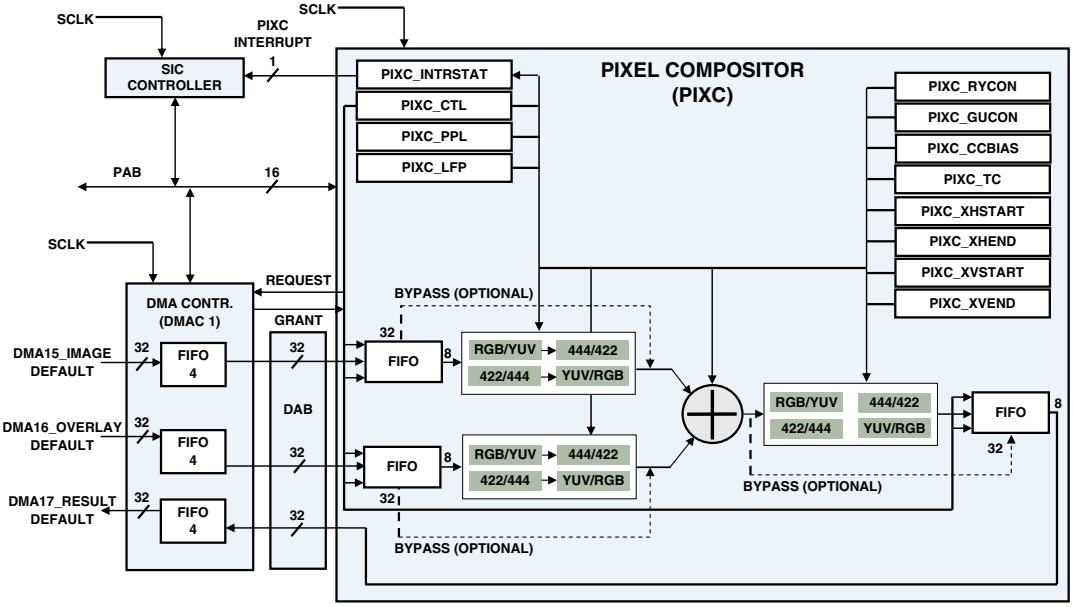


Figure 28-2. Pixel Compositor Functional Block Diagram

Description of Operation

This section describes the operation of the pixel compositor (PIXC).

General Description

The PIXC is used to combine and format the data streams required by a wide variety of digital LCD panels and NTSC/PAL analog video encoders. It provides all the control needed to allow two data streams from two separate data buffers to be combined and converted into appropriate formats for both LCD panels and video output displays. The main image buffer provides the basic background image presented in the data stream. The overlay image buffer allows the user to add foreground text and graphics

on top of the main image data stream. This feature is useful for printing additional graphical or textual information on the screen, such as symbols or a menu, while showing the main image in the background.

Overlay is an option and can be enabled or disabled. If it is disabled, the blender/compositor is bypassed and the data stream from the main image buffer goes directly to memory, with optional color space conversion.

Transparent color is just a special case of blending, masking off the blend operation on a pixel-by-pixel basis. In other words, if the overlay region consists of sub-regions that need to be transparent, this can be done by having these sub-regions in any particular color convenient to the programmer, and enabling the transparent color feature of the PIXC. Then, if the color data for a given overlay pixel matches the specified transparent color, the overlay function is masked for that pixel and its data is taken solely from the main image buffer, which is stored in memory in either YUV 4:2:2 interleaved format or RGB888 format.

Regardless of the data format or buffer structure, each color element is 8 bits wide. If overlay is enabled, a graphics/text overlay data buffer is defined in memory. The color space converter can switch positions among any of the three locations shown in Figure 28-1 on page 28-3; it can be in the image data path, the overlay data path, or after the blender. The exact position of the color space converter depends on the input and output data formats, which are discussed in more detail in the following sections.

Two dedicated DMA channels, with 32-bit bus widths, are used to transfer data from the main image data buffer and the overlay image data buffer into two separate PIXC FIFO buffers, where the data is then unpacked. Each of these FIFO buffers is 32 bits wide and contains 8 entries. The overlay data buffer can not be larger than the image buffer, and the overlay can be set to affect only selected portions of the main image. The position of the overlay in the main image is controlled by memory-mapped registers (MMRs) in the PIXC. In the blender, (8-bit) pixel elements from two buffers are mixed together. One dedicated DMA channel transfers the combined pixel data back to memory.

Since the end display may be a TV (NTSC/PAL) or an LCD panel, and since the image/overlay input buffers may be in either RGB888 or YUV 4:2:2 format, a color space conversion may be needed. The color space conversion is selected according to the input data stream format of the PIXC. A YUV-to-RGB format conversion is necessary if the end display is an LCD and if either of the PIXC input data streams is in YUV 4:2:2 format. Similarly, an RGB-to-YUV format conversion is necessary if the end display is a TV and if either of the PIXC input data streams is in RGB888 format.

If the final display device is an LCD, the output RGB data stream is always packed in RGB 8-bit serial format when transferring back to memory. Similarly, if the final display device is a TV, the YUV data stream is always packed in YUV 4:2:2 interleaved format when transferring back to memory.

Data Buffer Formats

For the implementation of overlay, the PIXC needs two input data streams from two separate data buffers, a main image buffer and an overlay buffer. The input data in these buffers must be in YUV 4:2:2 or RGB888 format (the main image data and the overlay data can be in different formats). The output data is also in one of these two formats, depending on the output display device being used.

Operation in YUV 4:2:2 Format

Each Y/U/V component is stored in 8 bits of data. The PIXC only accepts a YUV 4:2:2 interleaved format, in the following sequence:

V1, Y1, U1, Y2, V3, Y3, U3, Y4 …

(Two components with the same suffix number (for example, V1 and U1) implies they are extracted from the same pixel.)

It is the user's responsibility to ensure that the YUV source data to the PIXC is in the correct interleaved format. Therefore, data preprocessing may be necessary in order to meet this requirement.

Figure 28-3 and Figure 28-4 illustrate correct PIXC input buffer structure and data stream format.

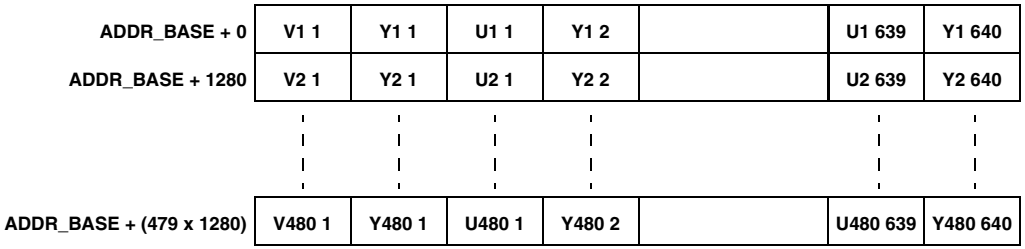


Figure 28-3. YUV 4:2:2 Data in an Interleaved Data Buffer Structure

Figure 28-4. YUV 4:2:2 Expected Data Stream Format to PIXC

The number of pixels per line in YUV mode must be an even number (for both input buffers), and the first chroma component in each line must be a V component.

Operation in RGB888 Format

Each R/G/B component is stored in 8 bits of data. Figure 28-5 and Figure 28-6 illustrate correct PIXC input buffer structure and data stream format.

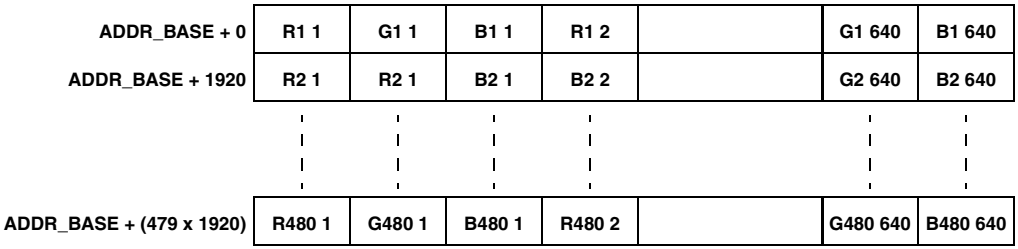


Figure 28-5. RGB888 Data Expected Data Buffer Structure

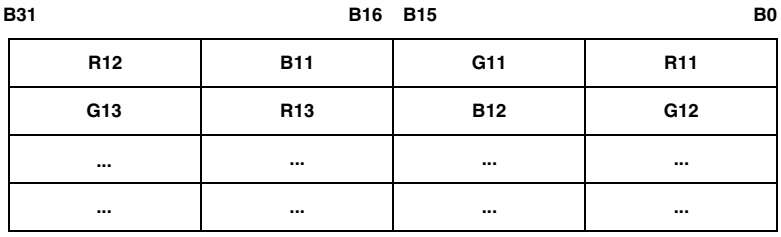


Figure 28-6. RGB888 Expected Data Stream Format to PIXC

For operation in RGB format, the total number of pixels in both input buffers must be a multiple of 4, so that the image boundary aligns with a 32-bit DMA word boundary.

DMA Channels

Three peripheral DMA channels can be assigned to the PIXC, as follows:

- A first input DMA channel is used for transferring either a part of the image or the entire main image data to the PIXC from memory (L3, L2, or L1).
- A second input DMA channel is used for transferring the overlay image to the PIXC from memory (L3, L2, or L1).
- An output DMA channel is used for transferring the blended data to memory (L3, L2 or L1).

For more information, see "Direct Memory Access" on page 7-1.

The two input DMA channels take the image data and overlay graphics/text data from their buffers into two separate FIFOs.

Whenever the PIXC is enabled, at least two DMA channels should be enabled and configured appropriately: the image DMA channel and the output DMA channel. Furthermore, when the overlay function of PIXC is enabled, the overlay DMA channel should be enabled and configured as well.

Functional Description

The PIXC implements the following main functions:

- Graphics/text overlay (including video overlay for small frame sizes)
- Transparency control (alpha blending) of the overlay pixel data
- Transparent color (chroma keying) of the overlay stream
- Color space conversion for LCD panels or NTSC/PAL displays

Data Overlay

Overlay is an optional function, so it can be enabled or disabled. If it is disabled, all overlay functionality is bypassed, and a single data stream from the main image data buffer goes directly to the image output buffer, after an optional format conversion. If it is enabled, the blender combines the pixel data from the two image input buffers.

The overlay image is located in a user-defined rectangle within the main image and, in most cases, the overlay image is smaller than the main image. Figure 28-7 illustrates an example of the main and overlay image regions on a screen, where a foreground triangle overlay sits on top of the main image in the background. Although the figure does not show this explicitly, (H-Start, V-Start) can equal (0,0).

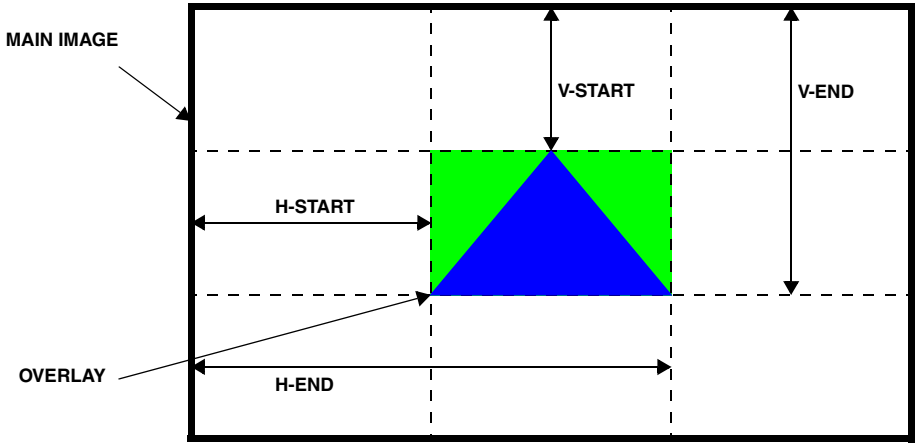


Figure 28-7. Main Image and Overlay Image Region

In certain situations, it may be beneficial to only DMA the region of the main image that is affected by the overlay instead of bringing in the entire main image. For example, in a situation where the intent is to overlay an image over the main image and to store the result back over the main

image, one could setup a 2D-DMA to only bring in the area of the main image that is affected by the overlay. This may reduce the amount of DMA activity thus potentially improving system performance.

There are two steps to implement the overlay process:

1. Defining an overlay buffer
    - The user must define a rectangular region that covers the whole overlay region no matter what shape the overlay content is. The overlay buffer holds the pixel data in the entire rectangular overlay region, which can include some areas where there is no overlay. In memory, these areas have to be filled with the transparent color value. (See "Transparency Control" on page 28-17.)
2. Configuring the overlay DMA
    - The user must define a single DMA descriptor for the overlay data transfer. The user must also fill the overlay coordinate registers in the PIXC with appropriate values. The overlay coordinate register set consists of two pairs of registers that specify the top left corner (H-Start, V-Start) and bottom right corner (H-End, V-End) of the overlay, along with a 4-bit register that specifies the α (transparency ratio) value. Each overlay is thus completely specified by a set of five registers. The widths and addresses of these registers are given in "PIXC Registers" on page 28-35.

There is a set of an additional five such registers that can be used to specify a second overlay region, so that two separate overlay blocks can be defined simultaneously. Furthermore, either or both of these overlay coordinate register sets can be enabled or disabled at one time, since separate enable bits (`OVR_A_EN` and `OVR_B_EN`) exist in the PIXC control register for each of the overlay register sets.

If there are more than two overlay blocks needed in a given application, the two sets of overlay registers must be managed by the user to perform the additional overlays. This can be done using an interrupt service routine, where the interrupt from the PIXC is used to re-program the overlay coordinate registers.

The PIXC can generate an interrupt under two conditions: at the end of the last valid overlay and at the end of a frame.

Either of these interrupts can be enabled or disabled. However, the PIXC only has one interrupt line output, so it raises an interrupt (under the appropriate condition) when either of these two interrupts is triggered. If both interrupts are enabled, the interrupt status register of the PIXC indicates which of the two conditions caused the interrupt to occur. Once the PIXC generates an interrupt, it stalls the pixel processing until software (ISR) clears the interrupt. However, the FIFOs do not stall and keep filling up even when the PIXC is in a stalled state. Both interrupts can be cleared by writing a 1 to the respective interrupt status bits.

After each interrupt (whether it is a last-valid-overlay interrupt or an end-of-frame interrupt), the PIXC restarts processing with coordinate register set A. In other words, at the time of clearing the interrupt:

- If coordinate set A is enabled (OVR_A_EN = 1), the PIXC assumes that the first incoming data over the DAB is to be overlaid on the area specified in coordinate set A.
- If coordinate set A is disabled (OVR_A_EN = 0), and coordinate set B is enabled (OVR_B_EN = 1), the PIXC assumes that the first incoming data over the DAB is to be overlaid on the area specified in coordinate set B.
- If both coordinate sets are disabled, the PIXC flushes the overlay FIFO and make no more data requests on the overlay DMA channel.

The overlay enable bits OVR_A_EN and OVR_B_EN should only be changed inside the interrupt service routines of the PIXC interrupts, or when the overlay block is disabled.

Note that the module enable bit (PIXC_EN) is the root enable for the PIXC. Both OVR_A_EN and OVR_B_EN are gated with PIXC_EN, so if PIXC_EN is set to zero, the individual overlay enable bits have no effect, and the module remains disabled. When PIXC_EN is programmed to zero, both the image and overlay FIFOs are flushed and no more DMA requests are made on either of the DMA channels.

Once the DMAs are enabled, the PIXC keeps track of the current pixel being displayed from the main image data by reading from two user-programmable registers: PIXC_PPL, which stores the number of pixels per line, and PIXC_LPF, which stores the number of lines per frame of the display device.

When the pixel count reaches the top left corner (H-Start, V-Start) of overlay data, the PIXC starts the overlay. When the pixel count reaches the top right corner (H-End, V-Start) of overlay data, the PIXC stops the

overlay. It starts again at the next line at (H-Start, V-Start+1) and stop at (H-End, V-Start + 1), and so on until the entire overlay frame is processed.

Internally, the start of the overlay DMA would have been pre-empted by the PIXC before the actual processing of the first overlay pixel, and DMA data would have been requested until the overlay FIFO were full. Similarly, the overlay DMA does not stop at the end of a line. The overlay FIFO continues to be filled with DMA data, even when the current pixel is not an overlay pixel, but the supply of overlay pixels from the overlay FIFO is simply halted.

The PIXC decides whether or not to perform overlay mixing for the current pixel by using the various PIXC register values. Therefore:

- The `PIXC_PPL` and `PIXC_LPF` must be programmed correctly (and cannot be 0).
- The `HSTART` and `HEND` must be less than or equal to `PIXC_PPL`.
- The `VSTART` and `VEND` must be less than or equal to `PIXC_LPF`.

The user can define multiple rectangular regions covering several separate overlays, using the same number of DMA descriptors, where each DMA descriptor corresponds to an overlay region.

Multiple overlay regions are split into two cases:

- Overlay regions with no horizontal overlap.

  This is a straightforward case (shown in Figure 28-8). Software can maintain separate areas in memory for both overlay regions, with separate H-Start, V-Start, H-End, and V-End coordinates for each

region. After the first overlay is completed, the DMA chain pointer can load the next overlay parameters (index, count, and modifier) to the DMA registers of the corresponding DMA channel.

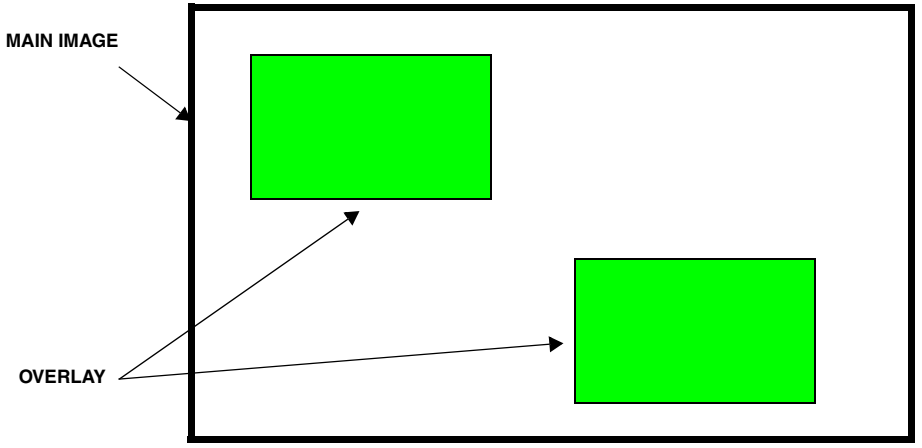


Figure 28-8. Overlay Regions With No Horizontal Overlap

- Overlay regions with horizontal overlap

  In these cases (see Figure 28-9), software has to maintain a combined overlay region in memory. This includes some in-between area where there is no overlay. This region of memory has to be filled with the transparent color value (explained below). The H-Start, V-Start, H-End and V-End coordinates contain the values of the combined overlay region.

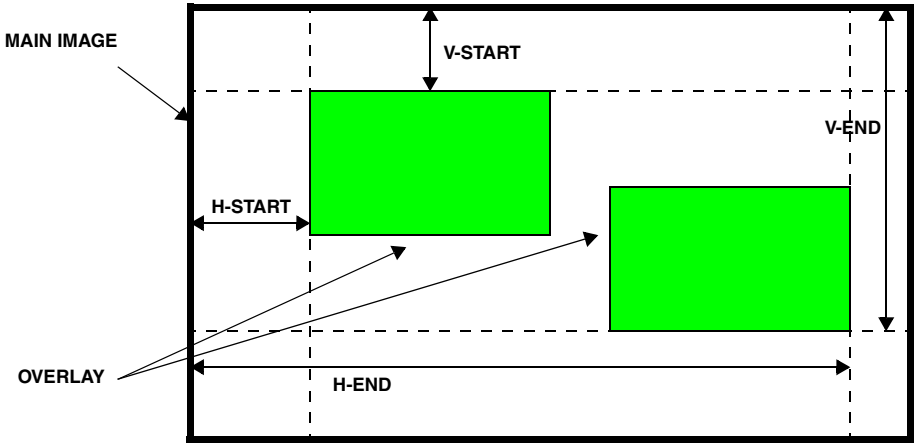


Figure 28-9. Overlay Regions With Horizontal Overlap

Transparency Control

When the overlay function is enabled, each overlay pixel is combined with each main image pixel to generate the displayed output pixel to be displayed. Each pixel combination is controlled by a transparency ratio value alpha (α), a 4-bit value that determines the proportion of overlay and main image that contribute to the output pixel. The pixel combination algorithm can be expressed as:

$$C = \frac{B(\alpha + 1)}{16} + \frac{A(15 - \alpha)}{16}$$

- A: 8-bit pixel data in main frame buffer (“background”)
- B: 8-bit pixel data in overlay buffer (“foreground”)
- C: 8-bit combined pixel data
- α: Transparency ratio code, which is a 4-bit value present in a memory-mapped register

Table 1 lists the multiplying factors for various α values.

Table 28-1. Multiplying Factors for Various α Values

α	Overlay Multiplying Factor	Image Multiplying Factor
0	1/16	15/16
1	2/16	14/16
2	3/16	13/16
3	4/16	12/16
4	5/16	11/16
5	6/16	10/16
6	7/16	9/16
7	8/16	8/16
8	9/16	7/16
9	10/16	6/16
10	11/16	5/16
11	12/16	4/16
12	13/16	3/16
13	14/16	2/16
14	15/16	1/16
15	1	0

Passing the image alone can be achieved by disabling the overlay function.

Rounding is performed at the output of the blender, which rounds the combined pixel data to the nearest integer value.

Transparent Color

A transparent color is a specific color that is removed from one image to reveal another “behind” it. This technique is also referred to as chroma keying. The principal subject is photographed or filmed against a background having a single color, usually in the blue or green spectrums. When the phase of the chroma signal corresponds to the pre-programmed state associated with the background color(s) behind the principal subject, the signal from the alternate background (which in this case comes from the main image channel) is inserted in the composite signal and presented at the output. When the phase of the chroma signal deviates from that associated with the background color(s) behind the principal subject, the picture data associated with the principal subject (in this case, the overlay image) is presented at the output. Figure 28-10 illustrates this concept.

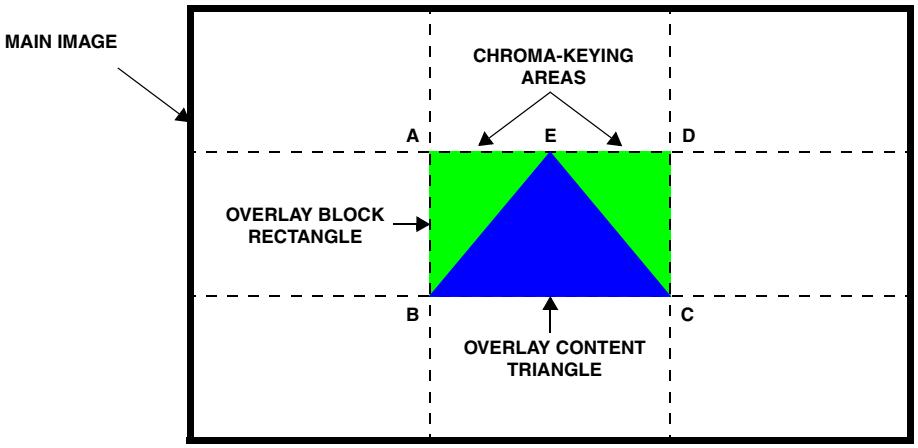


Figure 28-10. Transparent Color (Chroma Keying)

In order to display the main image in the two triangle areas ΔABE and ΔCDE in overlay block ABCD, the data in the overlay buffer corresponding to the pixels in the triangle areas ΔABE and ΔCDE must hold a specific value, called the transparent color.

The PIXC provides a 24-bit MMR (storing three 8-bit color components), for each of the two overlay blocks, in order to designate a particular RGB or YUV value as the transparent color. The transparent color must be in the same format (YUV 4:2:2 or RGB888) as the overlay data, regardless of whether or not a color space conversion is present in the overlay data path. The PIXC then compares each input pixel value on the overlay channel with this transparent color. If there is a match, the overlay pixel at this location is ignored by the blender, and the main image pixel at that location is assigned 100% weight.

If YUV 4:2:2 is the overlay channel input data format, artifacts may occur at the edge of the transparent color region. In this case, it is preferable to set the `UDS_MOD` bit to 0 (duplicating-dropping mode), in order to get better control of the U and V components at the edge of the transparent color region.

Color Space Conversion

As shown in Figure 28-1 on page 28-3, depending on the input data format and display device used, there may be a color space conversion performed on the data stream of the PIXC. If the input data is in YUV format, a YUV-to-RGB conversion can be performed for output to an LCD panel. If the input data is in RGB format, a RGB-to-YUV conversion can be performed for output to NTSC/PAL displays. The color space conversion may happen on any of the three paths (for example, the main image data path, the overlay image data path, or the combined data path). Register bits are used to specify the input, overlay and output formats.

The color space converter block has three main cases of operation:

1. Both the image and the overlay data are in the same format
2. The image and the overlay data are in different formats
3. Color space conversion only

These are all described in the following section, along with several special usage cases. Note that various scenarios may be shown in the same figure based on the output device chosen, though only a single output destination is supported at one time.

Case 1 - Image and Overlay in the Same Format

Both input data streams (main image and overlay) are in the same format, either YUV 4:2:2 or RGB888, so a color space conversion may be performed after alpha blending, depending on the output type. See Figure 28-11 and Figure 28-12.

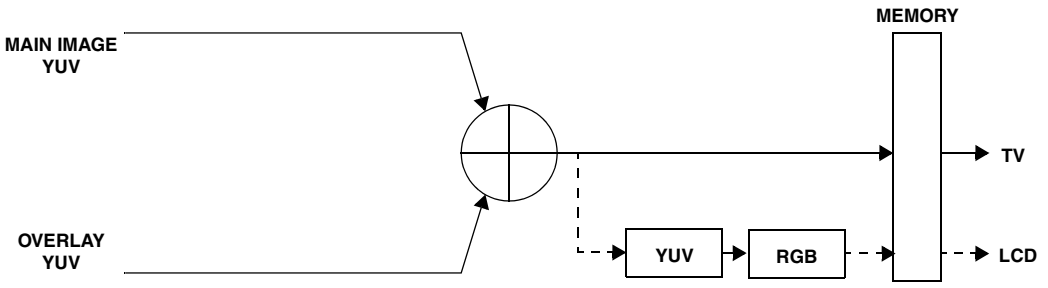


Figure 28-11. Both Input Data Streams in YUV 4:2:2 Format

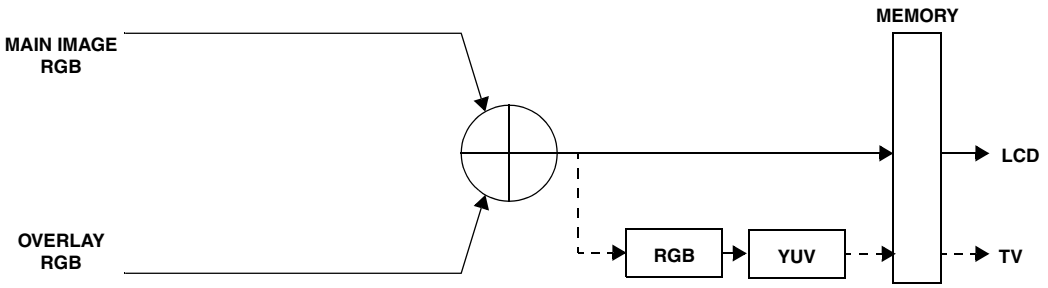


Figure 28-12. Both Input Data Streams in RGB888 Format

Case 2 - Image and Overlay in Different Formats

In this case, the two input data streams are not in the same format. The PIXC has to perform a color space conversion on either the main input stream or the overlay input stream (depending on the required output format) before alpha blending can take place. See Figure 28-13 and Figure 28-14.

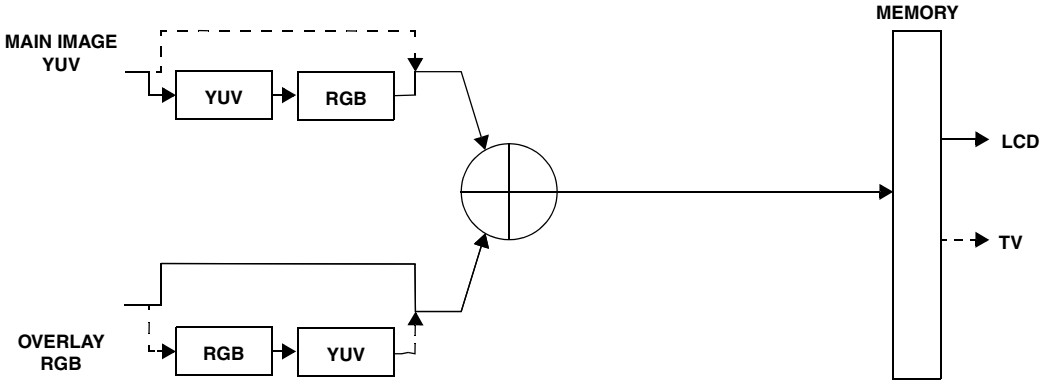


Figure 28-13. Main Image in YUV 4:2:2 and Overlay in RGB888

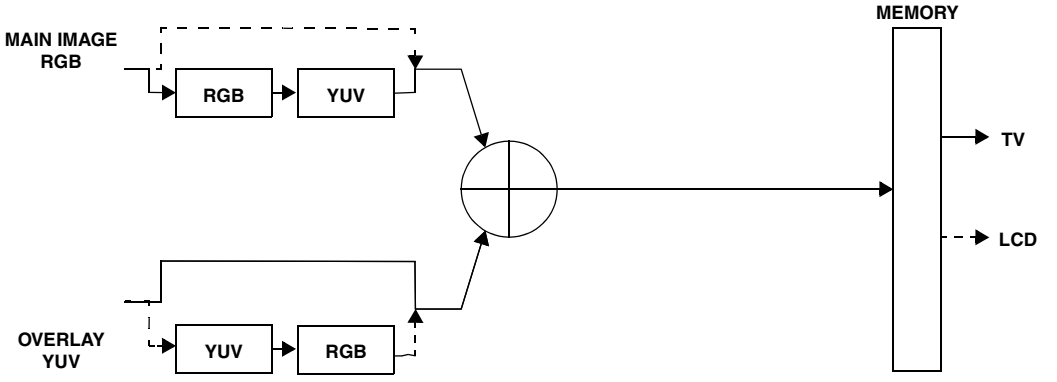


Figure 28-14. Main Image in RGB888 and Overlay in YUV 4:2:2

Case 3 - Color Space Conversion Only

In this case, there is no overlay blending. The main image is brought into the PIXC, the color space converted, and then sent back to memory. See Figure 28-15 and Figure 28-16.

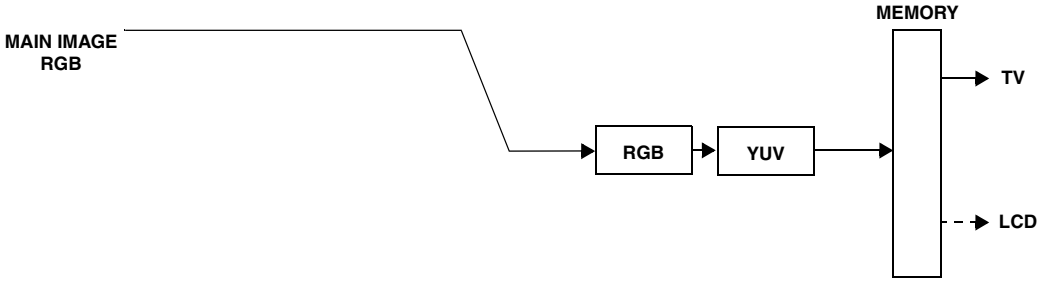


Figure 28-15. Main Image in RGB888 and Output in YUV 4:2:2 (No Overlay)

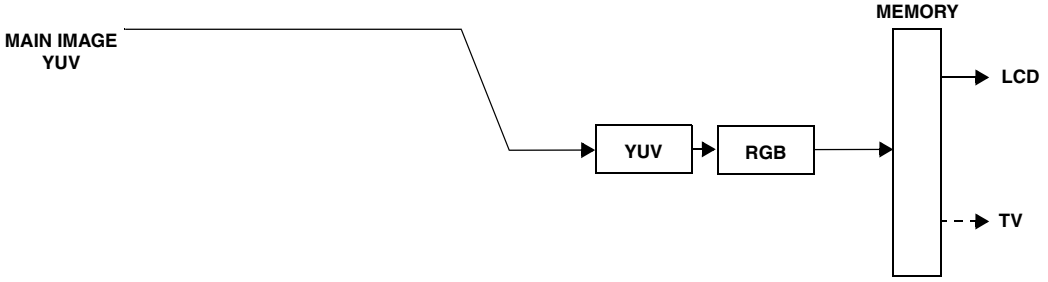


Figure 28-16. Main Image in YUV 4:2:2 and Output in RGB888 (No Overlay)

For this mode, the register settings are: `PIXC_EN` = 1, `OVR_A_EN` = 0 and `OVR_B_EN` = 0.

Color Space Conversion Matrix Equations

The PIXC color space conversion block implements the following matrix equation:

$$K \times \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} + \begin{bmatrix} A_{14} \\ A_{24} \\ A_{34} \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

The A_{xx} coefficients are 10-bit signed values represented in two's complement format. A_{11}…A_{33} are coefficient multipliers (for most cases, it is sufficient to specify these as integers between –512 and 511), and A_{14}, A_{24}, A_{34} are simply offsets added to the result for each row. B_1, B_2, B_3 represent the input pixel component values (for example, YUV or RGB) and C_1, C_2, C_3 are the output pixel component values. Output pixel values are rounded to the nearest integer.

The constant K equals 1/512. For example, to set A_{11}'s effective value to 0.299, this coefficient's MMR should be programmed to ROUND(.299*512), or 153. If a coefficient needs to be programmed with a value greater than 1, an extra bit exists in each coefficient's MMR to specify if an extra multiply by 4 must be performed after multiplying the input value by its coefficient. However, this setting can only be specified for an entire row, so if this bit is set, all the coefficients for that row (Ax1-Ax3) should be calculated as ROUND(coeff × 512/4). In other words, the constant K effectively becomes 1/128 for that row.

For reference, the matrix equations representing conversion between YUV and RGB formats are:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168 & -0.330 & 0.498 \\ 0.498 & -0.417 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.000 & 1.397 \\ 1.000 & -0.343 & -0.711 \\ 1.000 & 1.765 & 0.000 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} + \begin{bmatrix} -179 \\ 135 \\ -226 \end{bmatrix}$$

For YUV-to-RGB conversion, the PIXC expects the input data to be arranged in the following order: VYUY, VYUY, and so on (see Figure 28-3). As a result, if the input data is instead arranged as UYVY, UYVY, and so on, then, the columns Ax2 and Ax3 of the coefficient matrix are swapped.

For RGB-to-YUV conversion, the PIXC arranges the output data by default in the following order: VYUY, VYUY, and so on. If the output data is desired to instead be arranged as UYVY, UYVY, and so on, then, rows A2x and A3x of the coefficient and bias matrices are swapped.

Color Space Converter Output Thresholds

Each PIXC output sample is 8 bits wide, whether it is an R, G, B, Y, U or V component value. Therefore, any output sample must be in the 0 to 255 range. Since all the coefficients are programmable, some of the inputs, when operated upon by the coefficients, may produce an output outside the 0 to 255 range. In such cases, the PIXC clips the output component's value to 0 or 255.

YUV Conversion Modes

When the color space converter operates between two color spaces, it requires all components of each pixel to be present in the data stream. Therefore, the PIXC internally upsamples the YUV 4:2:2 data stream before a YUV-to-RGB conversion and similarly downsamples the YUV 4:4:4 data stream after a RGB-to-YUV conversion. The resampling always takes place between YUV 4:2:2 and YUV 4:4:4 formats, but a certain flexibility is provided with regard to how the resampling is done by the PIXC in each case.

Upsampling

A YUV 4:2:2-to-YUV 4:4:4 conversion can be performed either by averaging or by duplicating the pixel components. The UDS_MOD bit in the PIXC_CTL register specifies the upsampling mode. The default setting of this bit is 0, which corresponds to duplication of the chroma components (Us and Vs) from the odd pixels to the even pixels:

YUV 4:2:2 input:	V1Y1, U1Y2, V3Y3, U3Y4, …
YUV 4:4:4 conversion:	Y1U1V1, Y2U1V1, Y3U3V3, Y4U3V3, …

Setting the UDS_MOD bit to 1 enables the averaging of the chroma components of the preceding and succeeding pixels to obtain the intermediate chroma value. In other words, two consecutive odd-numbered pixels' chroma components are averaged to obtain the intermediate even-numbered pixel's chroma components:

YUV 4:2:2 input:	V1Y1, U1Y2, V3Y3, U3Y4, …
YUV 4:4:4 conversion:	Y1U1V1, Y2U2V2 [U2=(U1+U3)/2, V2=(V1+V3)/2],Y3U3V3, Y4U4V4 [U4=(U3+U5)/2, V4=(V3+V5)/2], …

If the sum of the preceding and succeeding pixels' U/V components is an odd number, the average is rounded down (truncated to an integer value).

Since the last pixel on a line is always an even-numbered pixel, the last odd pixel value on that line is used as the last even pixel value during upsampling.

Downsampling

A YUV 4:4:4-to-YUV 4:2:2 conversion can be performed either by averaging or by dropping the pixel components. The UDS_MOD bit also governs the downsampling mode. Setting the UDS_MOD bit to 0 (default) enables the dropping of the chroma components of the even numbered pixels:

YUV 4:4:4 input:	Y1U1V1, Y2U2V2, Y3U3V3, Y4U4V4, …
YUV 4:2:2 conversion:	V1Y1, U1Y2, V3Y3, U3Y4, …

Setting the UDS_MOD bit to 1 enables the averaging of the chroma components of two consecutive pixels to obtain a single chroma value for a pixel pair:

YUV 4:4:4 input:	Y1U1V1, Y2U2V2, Y3U3V3, Y4U4V4, …
YUV 4:2:2 conversion:	V12Y1, U12Y2, V34Y3, U34Y4, …
	[U12 = (U1+U2)/2, U34=(U3+U4)/2]
	[V12 = (V1+V2)/2, V34=(V3+V4)/2]

PIXC Actions

Table 2 lists the PIXC actions that take place based on any possible combination of image, overlay, and output data formats.

Table 28-2. PIXC Actions

Image Data format	Overlay Data format	Output Data format	PIXC Actions
YUV	No overlay	RGB	US followed by CSC
RGB	No overlay	YUV	CSC followed by DS
YUV	YUV	YUV	US in both paths followed by DS before output
YUV	RGB	RGB	US in image path, CSC in image path
YUV	YUV	RGB	US in both paths, followed by CSC
YUV	RGB	YUV	CSC in overlay path, US in image path, DS before output
RGB	YUV	YUV	CSC in image path, US in overlay path, DS before output
RGB	YUV	RGB	US in overlay path, CSC in overlay path
RGB	RGB	YUV	CSC followed by DS
RGB	RGB	RGB	No CSC, No US, No DS

- CSC = Color Space Conversion
- US = Upsampling
- DS = Downsampling
- YUV = YUV 4:2:2 format
- RGB = RGB888 format

Recommendations

For best results, the overlay should start on an odd-numbered pixel so that the U and V components of the image and the overlay are aligned. Otherwise artifacts may occur in the combined image.

When both the image and the overlay are in YUV 4:2:2 format and the output is also in YUV 4:2:2 format, the duplicating-dropping mode (`UDS_MOD`) is used to prevent a low-pass filtering effect on the images.

Special Usage Cases

There are ways by which the PIXC can be made to operate on certain data formats that it does not support in any standard modes. For example, YUV 4:4:4 is similar to RGB 888 with respect to the number of pixels per 32-bit DMA word. So the PIXC can be configured to work with the YUV 4:4:4 data format by intelligently programming the `IMG_FORM`, `OVR_FORM`, and `OUT_FORM` bit fields and the color space conversion coefficients.

These special usage cases are shown in Figure 28-17 through Figure 28-20.

Example 1 - Currently Defined Mode

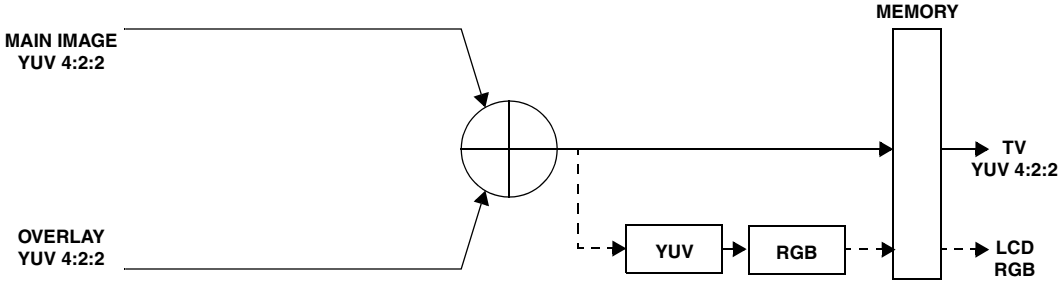


Figure 28-17. Example 1 - Currently Defined Mode

Example 1 - Special Usage of This Mode

- IMG_FORM = YUV
- OVR_FORM = YUV
- OUT_FORM = RGB
- All CSC coefficients = 1

In the special usage of this mode, YUV 4:2:2 inputs produce a blended YUV 4:4:4 data stream. A CSC matrix with coefficients of 1 is needed.

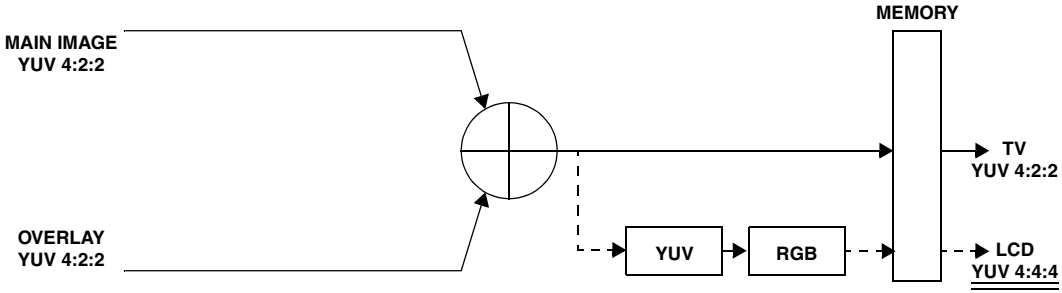


Figure 28-18. Example 1 - Special Usage

Example 2 - Currently Defined Mode

- IMG_FORM = RGB
- OVR_FORM = RGB
- OUT_FORM = YUV
- All CSC coefficients = 1

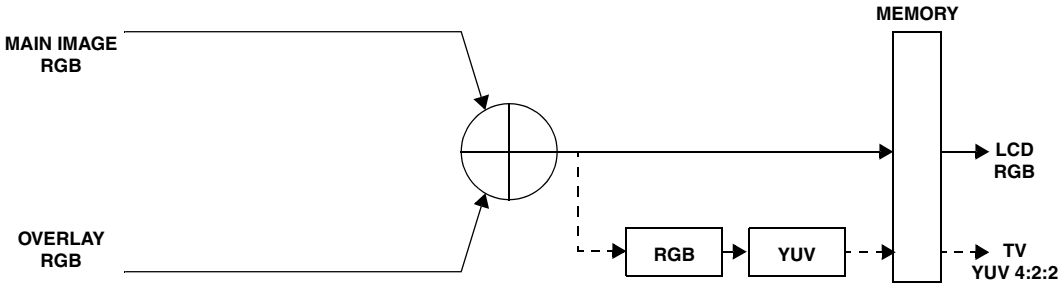


Figure 28-19. Example 2 - Currently Defined Mode

Example 2 - Special Usage of This Mode

In the special usage of this mode, YUV 4:4:4 input produces a blended YUV 4:2:2 or YUV 4:4:4 data stream. A CSC matrix with coefficients of 1 is needed.

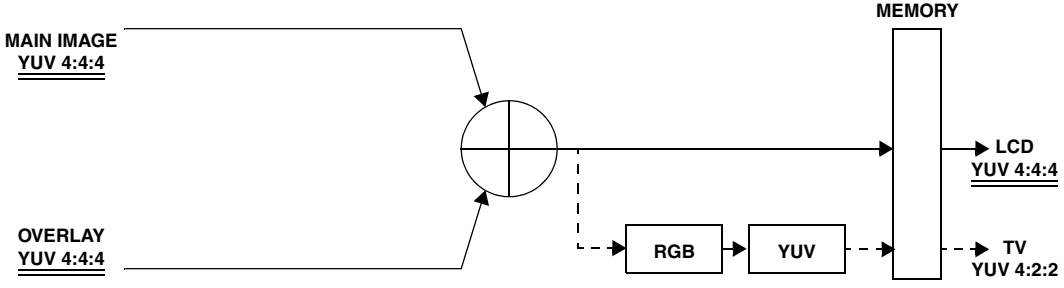


Figure 28-20. Example 2 - Special Usage

Example 3 - Currently Defined Mode

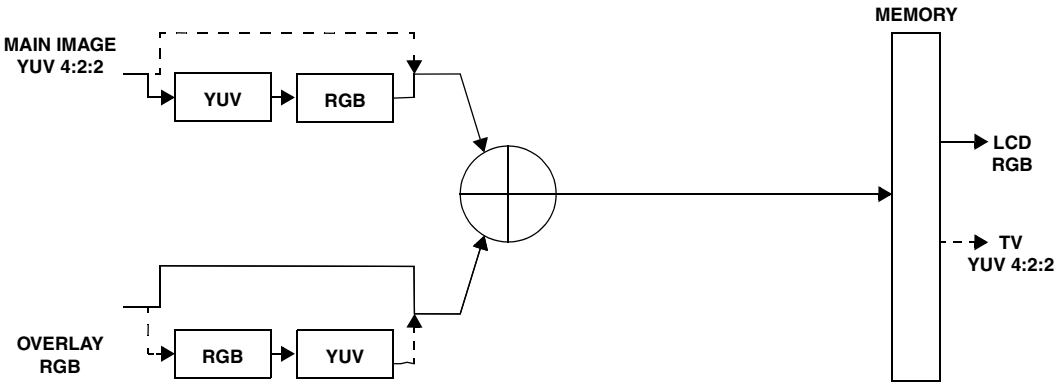


Figure 28-21. Example 3 - Currently Defined Mode

Example 3 - Special Usage of This Mode

In the special usage of this mode, a YUV 4:4:4 input stream and a YUV 4:2:2 input stream can be blended to produce either a YUV 4:4:4 or a YUV 4:2:2 output stream.

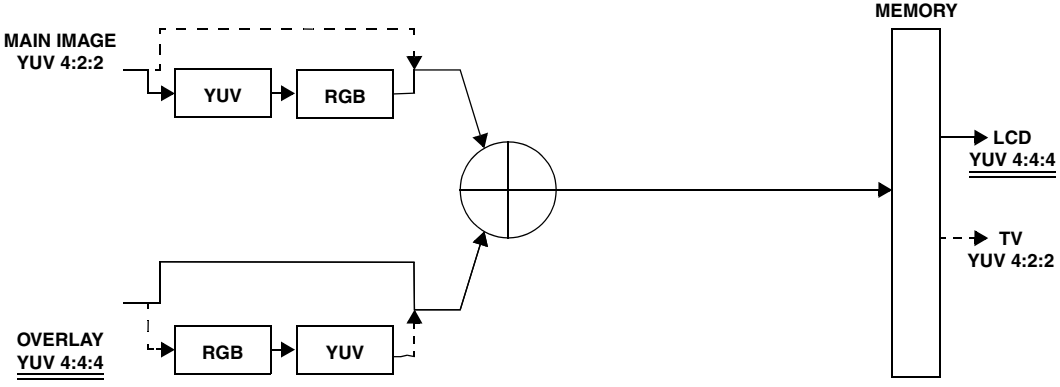


Figure 28-22. Example 3 - Special Usage

If the image format is YUV 4:2:2 and the overlay format is YUV 4:4:4:

- For YUV 4:2:2 output, program IMG_FORM = YUV, OVR_FORM = RGB, and OUT_FORM = YUV. Also program all the CSC coefficients to 1.
- For YUV 4:4:4 output, program IMG_FORM = YUV, OVR_FORM = RGB, and OUT_FORM = RGB. Also program all the CSC coefficients to 1.

If the image format is YUV4:4:4 and the overlay format is YUV 4:2:2, simply interchange IMG_FORM and OVR_FORM in the above programming cases.

Example 4 - Currently Defined Mode

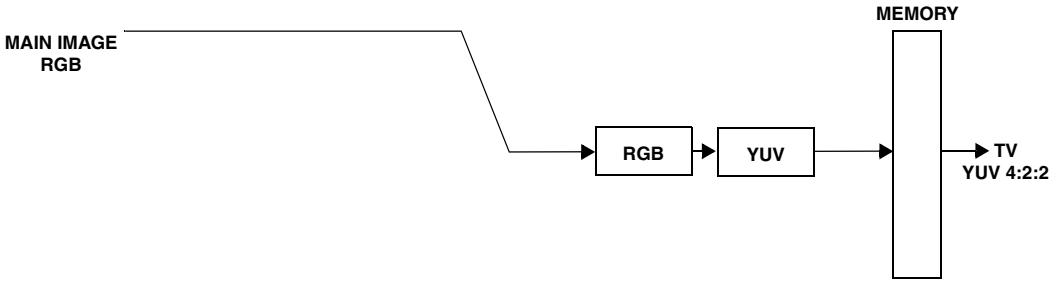


Figure 28-23. Example 4 - Currently Defined Mode

Example 4 - Special Usage of This Mode

- PIXC_EN = 1
- OVR_A_EN = 0
- OVR_B_EN = 0
- IMG_FORM = RGB

- OUT_FORM = YUV
- All CSC coefficients = 1

In the special usage of this mode, a simple downsampling from YUV 4:4:4 to YUV 4:2:2 is performed. Only color space conversion is enabled, using the PIXC_EN bit. A CSC matrix with coefficients of 1 is needed.

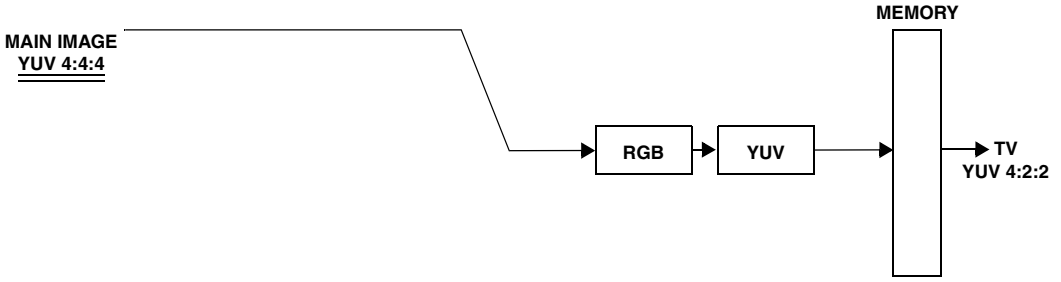


Figure 28-24. Example 4 - Special Usage

Programming Model

The following sections describe the PIXC programming model.

The output destination of the PIXC can be either an L3 frame buffer or an L2/L1 line buffer. As a recommendation for saving DMA bandwidth:

- If the size of the overlay content is relatively big, it is more efficient to send, through the DMA, the output of the PIXC to an L2/L1 line buffer, and then send the data from that line buffer directly through the EPPI to the display device.
- If the size of the overlay content is relatively small, it is more efficient to send, through the DMA, the output of the PIXC back to the L3 frame buffer, and then send the data from that frame buffer through the EPPI to the display device.

PIXC Registers

The PIXC has memory-mapped registers (MMRs) that regulate its operation. These registers are listed in Table 28-3. Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table 28-3. List of PIXC Memory-Mapped Registers

Register Name	Width	Address	Description
PIXC_CTL	16	0xFFC0 4400	"PIXC Control (PIXC_CTL) Register" on page 28-37
PIXC_PPL	16	0xFFC0 4404	"PIXC Pixels Per Line (PIXC_PPL) Register" on page 28-38
PIXC_LPF	16	0xFFC0 4408	"PIXC Lines Per Frame (PIXC_LPF) Register" on page 28-38
PIXC_AHSTART	16	0xFFC0 440C	Horizontal start pixel information of the overlay data (set A)."PIXC Horizontal Start (PIXC_xHSTART) Registers" on page 28-39
PIXC_AHEND	16	0xFFC0 4410	Horizontal end pixel information of the overlay data (set A)."PIXC Horizontal End (PIXC_xHEND) Registers" on page 28-39
PIXC_AVSTART	16	0xFFC0 4414	Vertical start pixel information of the overlay data (set A)."PIXC Vertical Start (PIXC_xVSTART) Registers" on page 28-40
PIXC_AVEND	16	0xFFC0 4418	Vertical end pixel information of the overlay data (set A)."PIXC Horizontal End (PIXC_xHEND) Registers" on page 28-39
PIXC_ATRANSP	16	0xFFC0 441C	Transparency ratio (set A)."PIXC Transparency Value (PIXC_xTRANSP) Registers" on page 28-41
PIXC_BHSTART	16	0xFFC0 4420	Horizontal start pixel information of the overlay data (set B)."PIXC Horizontal Start (PIXC_xHSTART) Registers" on page 28-39
PIXC_BHEND	16	0xFFC0 4424	Horizontal end pixel information of the overlay data (set B)."PIXC Horizontal End (PIXC_xHEND) Registers" on page 28-39

Table 28-3. List of PIXC Memory-Mapped Registers (Cont'd)

Register Name	Width	Address	Description
PIXC_BVSTART	16	0xFFC0 4428	Vertical start pixel information of the overlay data (set B)."PIXC Vertical Start (PIXC_xVSTART) Registers" on page 28-40
PIXC_BVEND	16	0xFFC0 442C	Vertical end pixel information of the overlay data (set B)."PIXC Vertical End (PIXC_xVEND) Registers" on page 28-40
PIXC_BTRANSP	16	0xFFC0 4430	Transparency ratio (set B)."PIXC Transparency Value (PIXC_xTRANSP) Registers" on page 28-41
PIXC_INTRSTAT	16	0xFFC0 443C	"PIXC Interrupt Status (PIXC_INTRSTAT) Register" on page 28-41
PIXC_RYCON	32	0xFFC0 4440	R/Y conversion coefficients."PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register" on page 28-42
PIXC_GUCON	32	0xFFC0 4444	G/U conversion coefficients."PIXC G/U Conversion Coefficient (PIXC_GUCON) Register" on page 28-43
PIXC_BVCON	32	0xFFC0 4448	B/V conversion coefficients."PIXC B/V Conversion Coefficient (PIXC_BVCON) Register" on page 28-44
PIXC_CCBIAS	32	0xFFC0 444C	"PIXC Color Conversion Bias (PIXC_CCBIAS) Register" on page 28-45
PIXC_TC	32	0xFFC0 4450	"PIXC Transparency Color Value (PIXC_TC) Register" on page 28-46

All PIXC registers have a default value of zero, except the transparency ratio registers which have a default value of 0xF.

The programmer should avoid writing to any of the MMRs when the module is enabled. Writing to the MMRs during the module enabled state can lead to unpredictable behavior of the PIXC. All MMRs can be read when the PIXC is in the enabled state, and this does not cause any change of status in the PIXC, but register writes should happen only when the PIXC is disabled, or stalled by an interrupt condition.

The following sections provide bit descriptions of the PIXC registers.

PIXC Control (PIXC_CTL) Register

The PIXC_CTL register (Figure 28-25) provides overlay enable, resampling mode selection, input/output data format selection, transparent color enable, watermark level selection, and image/overlay FIFO status.

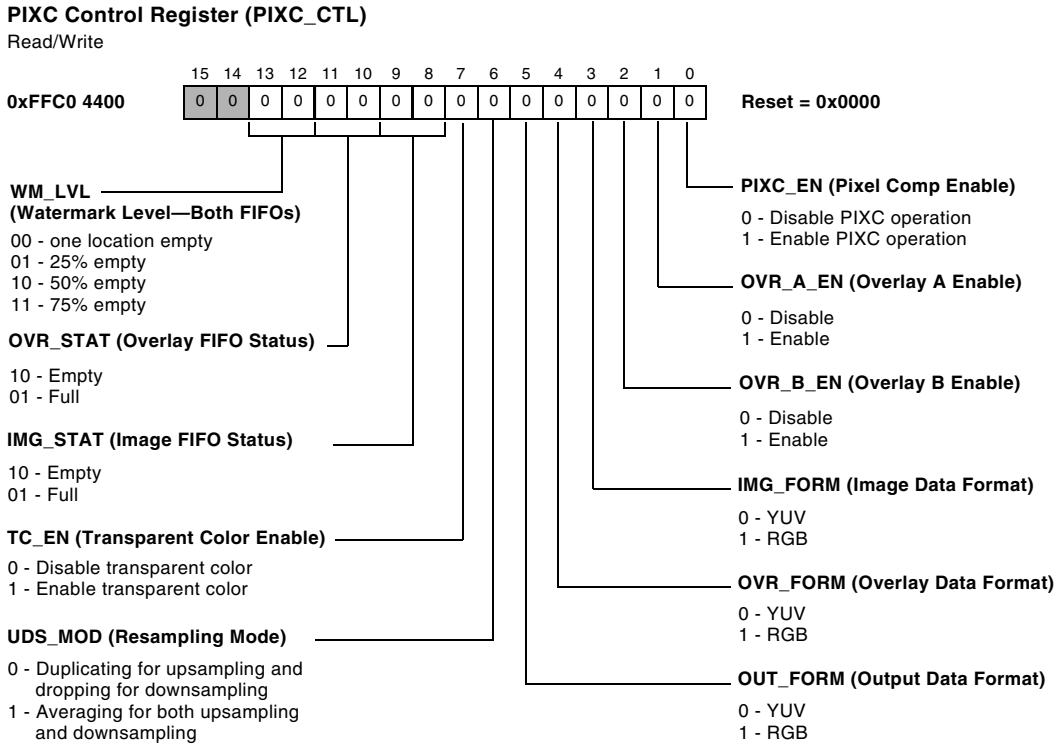


Figure 28-25. PIXC Control Register

Watermarking is described in the C/C++ Compiler and Library Manual for BLACKfin DSPs.

PIXC Pixels Per Line (PIXC_PPL) Register

The PIXC_PPL register (Figure 28-26) provides the number of pixels per line of the display.

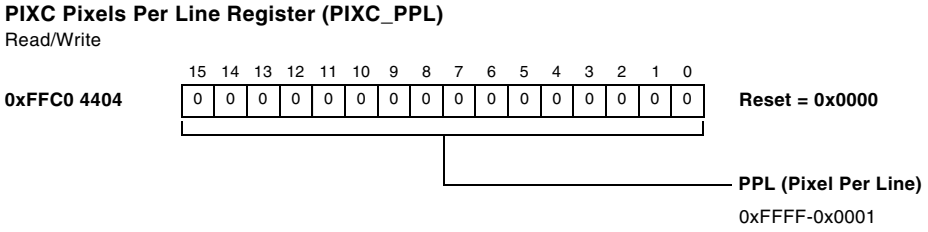


Figure 28-26. PIXC Pixels Per Line Register

PIXC Lines Per Frame (PIXC_LPF) Register

The PIXC_LPF register (Figure 28-26) provides the number of lines per frame of the display.

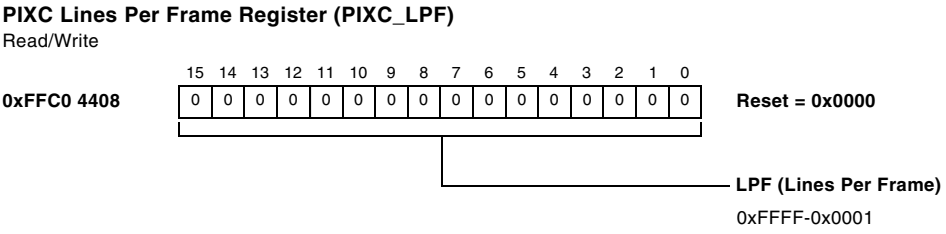


Figure 28-27. PIXC Lines Per Frame Register

PIXC Horizontal Start (PIXC_xHSTART) Registers

The PIXC_AHSTART and PIXC_BHSTART registers (Figure 28-28) provide the horizontal start pixel coordinates of the overlay data.

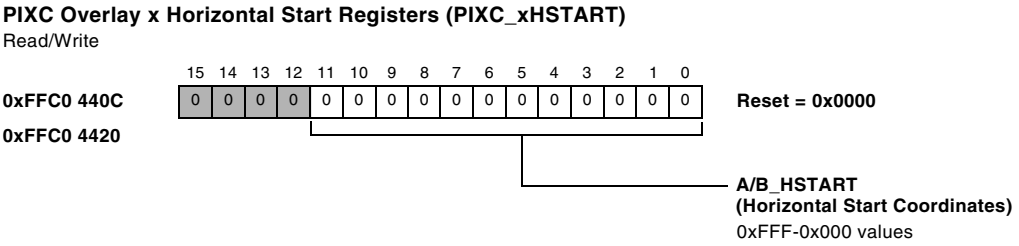


Figure 28-28. PIXC Horizontal Start Registers

PIXC Horizontal End (PIXC_xHEND) Registers

The PIXC_AHEND and PIXC_BHEND registers (Figure 28-29) provide the horizontal end pixel coordinates of the overlay data.

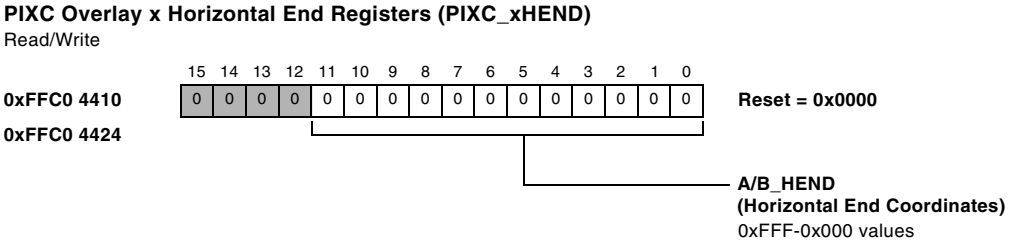


Figure 28-29. PIXC Horizontal End Registers

PIXC Vertical Start (PIXC_xVSTART) Registers

The PIXC_AVSTART and PIXC_BVSTART registers (Figure 28-30) provide the vertical start pixel coordinates of the overlay data.

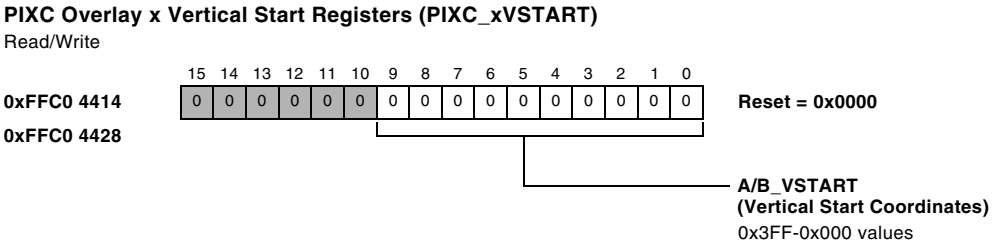


Figure 28-30. PIXC Vertical Start Registers

PIXC Vertical End (PIXC_xVEND) Registers

The PIXC_AVEND and PIXC_BVEND registers (Figure 28-31) provide the vertical end pixel coordinates of the overlay data.

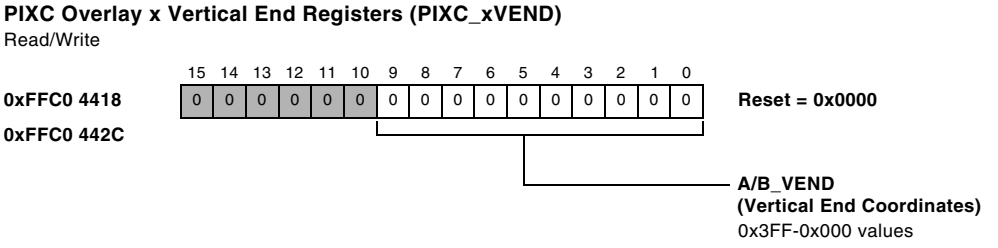


Figure 28-31. PIXC Vertical End Registers

PIXC Transparency Value (PIXC_xTRANSP) Registers

The PIXC_ATRANSP and PIXC_BTRANSP registers (Figure 28-32) provide the overlay transparency ratio values.

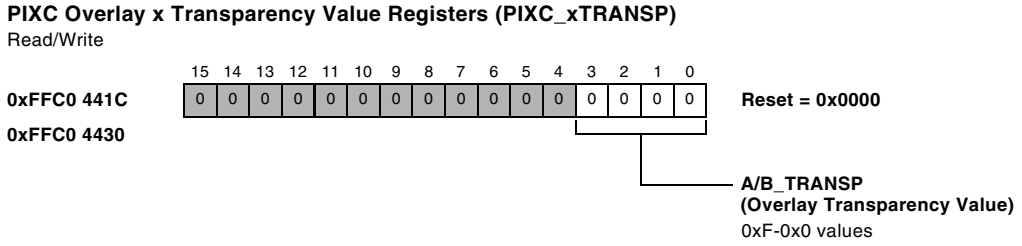


Figure 28-32. PIXC Transparency Value Registers

PIXC Interrupt Status (PIXC_INTRSTAT) Register

The PIXC_INTRSTAT register (Figure 28-33) provides overlay interrupt configuration and status information.

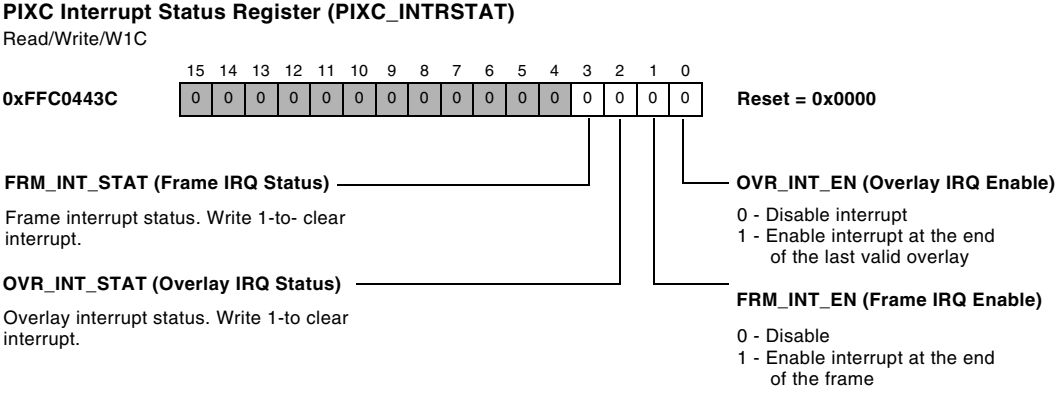


Figure 28-33. PIXC Interrupt Status Register

PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register

The PIXC_RYCON register (Figure 28-34) provides the R/Y conversion coefficients in the color space conversion matrix.

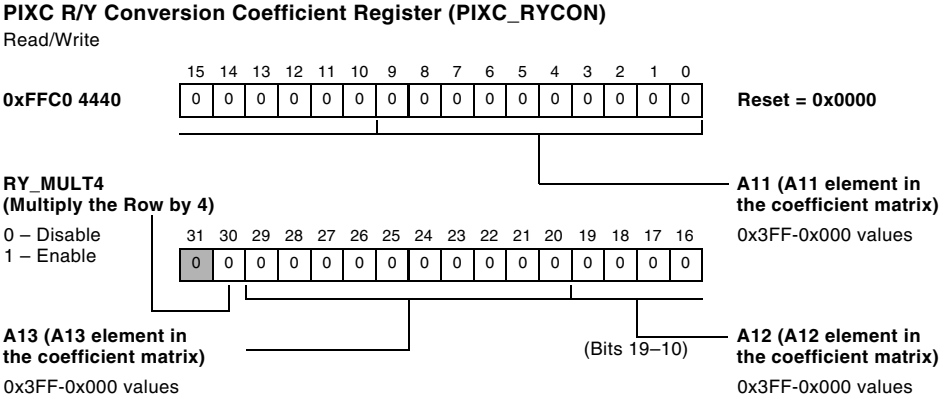


Figure 28-34. PIXC R/Y Conversion Coefficient Register

PIXC G/U Conversion Coefficient (PIXC_GUCON) Register

The PIXC_GUCON register (Figure 28-35) provides the G/U conversion coefficients in the color space conversion matrix.

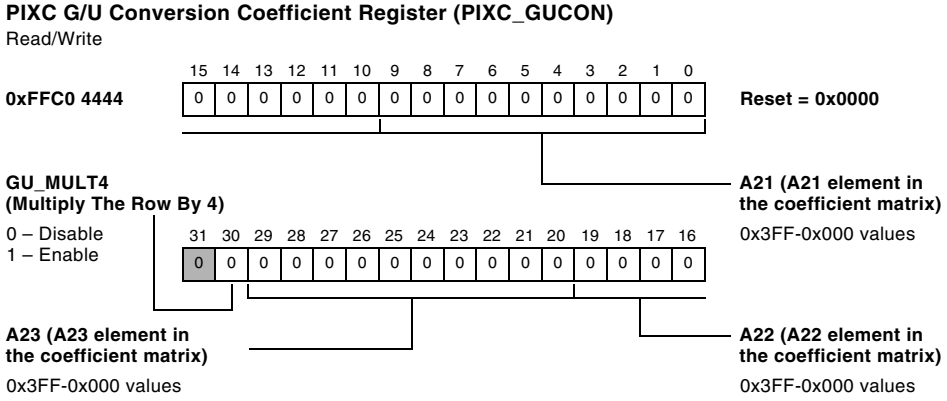


Figure 28-35. PIXC G/U Conversion Coefficient Register

PIXC B/V Conversion Coefficient (PIXC_BVCON) Register

The PIXC_BVCON register (Figure 28-36) provides the B/V conversion coefficients in the color space conversion matrix.

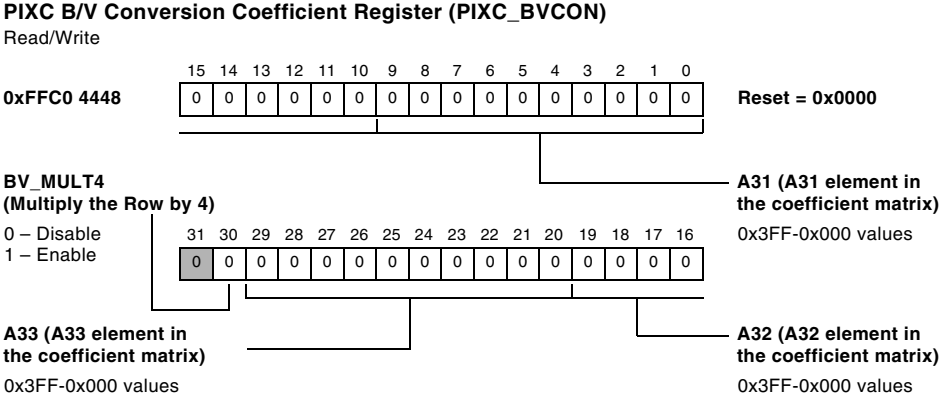


Figure 28-36. PIXC B/V Conversion Coefficient Register

PIXC Color Conversion Bias (PIXC_CCBIAS) Register

The PIXC_CCBIAS register (Figure 28-37) provides the bias values in the color space conversion matrix.

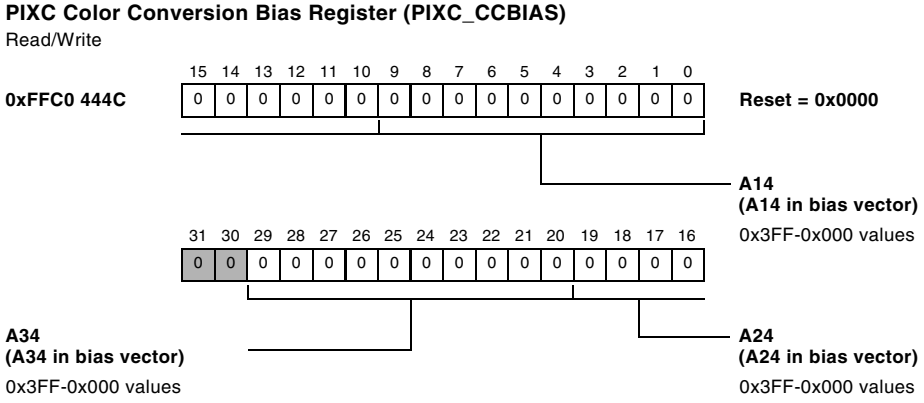


Figure 28-37. PIXC Color Conversion Bias Register

PIXC Transparency Color Value (PIXC_TC) Register

The PIXC_TC register (Figure 28-38) provides the transparent color value.

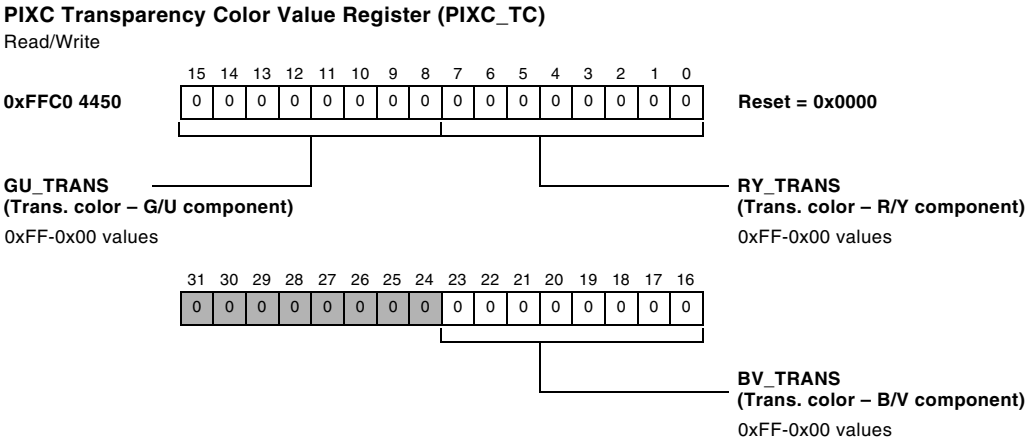


Figure 28-38. PIXC Transparency Color Value Register

29 MEDIA TRANSCEIVER MODULE (MXVR)

Among the ADSP-BF54x processor Blackfin processors, the MXVR is only available on the ADSP-BF549 processor.

This chapter includes the following sections:

- "Overview" on page 29-1
- "Interface Signals" on page 29-2
- "MXVR Memory Map" on page 29-4
- "MXVR Registers" on page 29-4
- "General Operation" on page 29-108

Overview

The media transceiver module (MXVR) serves as the network interface to a Media-Oriented System Transport (MOST®) ring network for the ADSP-BF54x processor Blackfin processor. The MXVR can be directly connected to an optical PHY. The optical PHY, the MXVR module and the MXVR device driver (Network Services Layer 1) together implement the MOST® NetInterface.

The MXVR is capable of transmitting and receiving synchronous data streams, asynchronous packet data, and control messages on the MOST® bus. The MXVR is fully compatible with industry standard MOST® network transceiver devices. The MXVR can simultaneously transmit and

receive the full bandwidth of the bus (24M bps). The MXVR offers fast lock times, greater jitter immunity, and a sophisticated DMA scheme for data transfers.

The DMA capabilities of the MXVR make transmission and reception of data (synchronous data, asynchronous packets, and control messages) easy. All data to be transmitted and all data received is stored in L1 memory. This gives the ADSP-BF54x processor core fast and easy access to the data. Data is transferred from L1 memory to the MXVR for transmission on the MOST® bus and data received from the MOST® bus by the MXVR is DMA'ed into L1 memory.

The MXVR has 14 dedicated DMA channels that work autonomously from the ADSP-BF54x processor processor core. Synchronous data can be transferred to and from synchronous channels through eight synchronous data DMA channels. Two more DMA channels support the transmission and reception of asynchronous packet data and another two DMA channels support the transmission and reception of control messages. Also, two additional DMA channels support Remote Read and Remote Write control messages.

The MXVR can act as the network master or as a network slave in a MOST® network containing other ADSP-BF54x processor nodes or other MOST® transceivers.

Interface Signals

Table 29-1 lists the MXVR signal pins. All output pins are 3.3V compliant with the exception of $\overline{\texttt{MTXON}}$ which can also be configured as an open drain output and can be pulled up to 5V. The MRX and $\overline{\texttt{MRXON}}$ input pins are 5V tolerant. All signal pins except for the dedicated crystal oscillator pins MXI and MXO, the MFS output pin, and the analog MLF_P and MLF_M pin are multiplexed with GPIO and other peripheral functions. The selection

of whether the pin has the MXVR functionality or has GPIO or other peripheral functionality is set within the ADSP-BF54x processor GPIO module. For more information see Chapter 9, “General-Purpose Ports”.

Table 29-1. MXVR Signal Pins

Pin Name	MXVR Signal Name	MXVR Signal Function	MXVR Signal Direction
MXI	MXI	MXVR Crystal Input	Input
MXO	MXO	MXVR Crystal Output	Output
PH6	MRX	MXVR Receive Data	Input (5V tolerant)
PH5	MTX	MXVR Transmit Data	Output
PC1	MMCLK	MXVR Master Clock	Output
PC5	MBCLK	MXVR Bit Clock	Output
MFS	MFS	MXVR Frame Sync	Output
PG11	$\overline{\text{MTXON}}$	MXVR Transmit PHY On	Output (5V tolerant)
PH7	$\overline{\text{MRXON}}$	MXVR Receive PHY On	Input (5V tolerant)
MLF_P	MLF_P	MXVR Loop Filter Plus	Analog
MLF_M	MLF_M	MXVR Loop Filter Minus	Analog

Table 29-2 lists the special power and ground pins needed for the MXVR. These supply pins are routed out to signal pins on the package for noise isolation.

Table 29-2. MXVR Supply Pins

Signal Name	Function	Supply
VDDMC	MXVR Crystal Power Supply	3.3 V
GNDMC	MXVR Crystal Ground	Ground
VDDMX	MXVR I/O Power Supply	3.3 V
GNDMX	MXVR I/O Ground	Ground
VDDMP	MXVR PLL Power Supply	1.2 V
GNDMP	MXVR PLL Ground	Ground

MXVR Memory Map

Table A-18 on page A-24 shows the memory map for the MXVR. All MXVR MMRs appear on the PAB bus. All MMR addresses are aligned to 32-bit address boundaries. An incorrectly sized or misaligned read to an MMR generates a bus error exception and the data returned will be unknown. An incorrectly sized or misaligned write to an MMR generates a hardware error interrupt and the write does not modify the MMR.

MXVR Registers

Table 29-3 lists the MXVR registers.

Table 29-3. MXVR Registers

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2700	MXVR_CONFIG	"MXVR Configuration (MXVR_CONFIG) Register" on page 29-13	16 R/W	0x1FCA
0xFFC0 2704	Reserved	–	–	–

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2708 0xFFC0 270C	MXVR_STATE_0 MXVR_STATE_1	"MXVR State (MXVR_STATE_0, MXVR_STATE_1) Registers" on page 29-19	32 RO	0x0000 0000
0xFFC0 2710 0xFFC0 2714	MXVR_INT_STAT_0 MXVR_INT_STAT_1	"MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)" on page 29-29	32 R/W	0x0000 0000
0xFFC0 2718 0xFFC0 271C	MXVR_INT_EN_0 MXVR_INT_EN_1	"MXVR Interrupt Enable 0 (MXVR_INT_EN_0) Register" on page 29-43	32 R/W	0x0000 0000
0xFFC0 2720	MXVR_POSITION	"MXVR Node Position (MXVR_POSITION) Register" on page 29-48	16 RO	0x8000
0xFFC0 2724	MXVR_MAX_POSITION	"MXVR Maximum Node Position (MXVR_MAX_POSITION) Register" on page 29-49	16 RO	0x0000
0xFFC0 2728	MXVR_DELAY	"MXVR Node Frame Delay (MXVR_DELAY) Register" on page 29-50	16 RO	0x8000
0xFFC0 272C	MXVR_MAX_DELAY	"MXVR Maximum Node Frame Delay (MXVR_MAX_DELAY) Register" on page 29-52	16 RO	0x0000
0xFFC0 2730	MXVR_LADDR	"MXVR Logical Address (MXVR_LADDR) Register" on page 29-53	32 R/W	0x0000 FFFF

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2734	MXVR_GADDR	"MXVR Group Address (MXVR_GADDR) Register" on page 29-54	16 R/W	0x0000
0xFFC0 2738	MXVR_AADDR	"MXVR Alternate Address (MXVR_AADDR) Register" on page 29-55	32 R/W	0x0000 0FFF
0xFFC0 273C 0xFFC0 2740 0xFFC0 2744 0xFFC0 2748 0xFFC0 274C 0xFFC0 2750 0xFFC0 2754 0xFFC0 2758 0xFFC0 275C 0xFFC0 2760 0xFFC0 2764 0xFFC0 2768 0xFFC0 276C 0xFFC0 2770 0xFFC0 2774	MXVR_ALLOC_0 MXVR_ALLOC_1 MXVR_ALLOC_2 MXVR_ALLOC_3 MXVR_ALLOC_4 MXVR_ALLOC_5 MXVR_ALLOC_6 MXVR_ALLOC_7 MXVR_ALLOC_8 MXVR_ALLOC_9 MXVR_ALLOC_10 MXVR_ALLOC_11 MXVR_ALLOC_12 MXVR_ALLOC_13 MXVR_ALLOC_14	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55	32 RO	0xXXXX XXXX
0xFFC0 2778 0xFFC0 277C 0xFFC0 2780 0xFFC0 2784 0xFFC0 2788 0xFFC0 278C 0xFFC0 2790 0xFFC0 2794	MXVR_SYNC_LCHAN_0 MXVR_SYNC_LCHAN_1 MXVR_SYNC_LCHAN_2 MXVR_SYNC_LCHAN_3 MXVR_SYNC_LCHAN_4 MXVR_SYNC_LCHAN_5 MXVR_SYNC_LCHAN_6 MXVR_SYNC_LCHAN_7	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57	32 R/W	0xFFFF FFFF

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2798 0xFFC0 27AC 0xFFC0 27C0 0xFFC0 27D4 0xFFC0 27E8 0xFFC0 27FC 0xFFC0 2810 0xFFC0 2824	MXVR_DMA0_CONFIG MXVR_DMA1_CONFIG MXVR_DMA2_CONFIG MXVR_DMA3_CONFIG MXVR_DMA4_CONFIG MXVR_DMA5_CONFIG MXVR_DMA6_CONFIG MXVR_DMA7_CONFIG	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59	32 R/W	0x0000 0000
0xFFC0 279C 0xFFC0 27B0 0xFFC0 27C4 0xFFC0 27D8 0xFFC0 27EC 0xFFC0 2800 0xFFC0 2814 0xFFC0 2828	MXVR_DMA0_START_ADDR MXVR_DMA1_START_ADDR MXVR_DMA2_START_ADDR MXVR_DMA3_START_ADDR MXVR_DMA4_START_ADDR MXVR_DMA5_START_ADDR MXVR_DMA6_START_ADDR MXVR_DMA7_START_ADDR	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69	32 R/W	0xFF00 0000
0xFFC0 27A0 0xFFC0 27B4 0xFFC0 27C8 0xFFC0 27DC 0xFFC0 27F0 0xFFC0 2804 0xFFC0 2718 0xFFC0 272C	MXVR_DMA0_COUNT MXVR_DMA1_COUNT MXVR_DMA2_COUNT MXVR_DMA3_COUNT MXVR_DMA4_COUNT MXVR_DMA5_COUNT MXVR_DMA6_COUNT MXVR_DMA7_COUNT	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72	16 R/W	0x0001
0xFFC0 27A4 0xFFC0 27B8 0xFFC0 27CC 0xFFC0 27E0 0xFFC0 27F4 0xFFC0 2808 0xFFC0 281C 0xFFC0 2830	MXVR_DMA0_CURR_ADDR MXVR_DMA1_CURR_ADDR MXVR_DMA2_CURR_ADDR MXVR_DMA3_CURR_ADDR MXVR_DMA4_CURR_ADDR MXVR_DMA5_CURR_ADDR MXVR_DMA6_CURR_ADDR MXVR_DMA7_CURR_ADDR	"MXVR DMA Channel x Current Address (MXVR_DMA0_CUR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71	32 RO	0xFF00 0000

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 27A8 0xFFC0 27BC 0xFFC0 27D0 0xFFC0 27E4 0xFFC0 27F8 0xFFC0 280C 0xFFC0 2820 0xFFC0 2834	MXVR_DMA0_CURR_COUNT MXVR_DMA1_CURR_COUNT MXVR_DMA2_CURR_COUNT MXVR_DMA3_CURR_COUNT MXVR_DMA4_CURR_COUNT MXVR_DMA5_CURR_COUNT MXVR_DMA6_CURR_COUNT MXVR_DMA7_CURR_COUNT	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74	16 RO	0x0000
0xFFC0 2838	MXVR_AP_CTL	"MXVR Asynchronous Packet Control (MXVR_AP_CTL) Register" on page 29-75	16 R/W	0x0000
0xFFC0 283C	MXVR_APRB_START_ADDR	"MXVR Asynchronous Packet Receive Buffer Start Address (MXVR_APRB_START_ADDR) Register" on page 29-77	32 R/W	0xFF00 0000
0xFFC0 2840	MXVR_APRB_CURR_ADDR	"MXVR Asynchronous Packet Receive Buffer Current Address (MXVR_APRB_CURR_ADDR) Register" on page 29-78	32 RO	0xFF00 0000
0xFFC0 2844	MXVR_APTB_START_ADDR	"MXVR Asynchronous Packet Transmit Buffer Start Address (MXVR_APTB_START_ADDR) Register" on page 29-79	32 R/W	0xFF00 0000
0xFFC0 2848	MXVR_APTB_CURR_ADDR	"MXVR Asynchronous Packet Transmit Buffer Current Address (MXVR_APTB_CURR_ADDR) Register" on page 29-79	32 RO	0xFF00 0000

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 284C	MXVR_CM_CTL	"MXVR Control Message Control (MXVR_CM_CTL) Register" on page 29-80	32 R/W	0x0000 0000
0xFFC0 2850	MXVR_CMRB_START_ADDR	"MXVR Control Message Receive Buffer Start Address (MXVR_CMRB_START_ADDR) Register" on page 29-82	32 R/W	0xFF00 0000
0xFFC0 2854	MXVR_CMRB_CURR_ADDR	"MXVR Control Message Receive Buffer Current Address (MXVR_CMRB_CURR_ADDR) Register" on page 29-83	32 RO	0xFF00 0000
0xFFC0 2858	MXVR_CMTB_START_ADDR	"MXVR Control Message Transmit Buffer Start Address (MXVR_CMTB_START_ADDR) Register" on page 29-84	32 R/W	0xFF00 0000
0xFFC0 285C	MXVR_CMTB_CURR_ADDR	"MXVR Control Message Transmit Buffer Current Address (MXVR_CMTB_CURR_ADDR) Register" on page 29-85	32 RO	0xFF00 0000
0xFFC0 2860	MXVR_RRDB_START_ADDR	"MXVR Remote Read Buffer Start Address (MXVR_RRDB_START_ADDR) Register" on page 29-86	32 R/W	0xFF00 0000

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2864	MXVR_RRDB_CURR_ADDR	"MXVR Remote Read Buffer Current Address (MXVR_RRDB_CURR_ADDR) Register" on page 29-86	32 RO	0xFF00 0000
0xFFC0 2868 0xFFC0 2870	MXVR_PAT_DATA_0 MXVR_PAT_DATA_1	"MXVR Pattern Data (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1) Registers" on page 29-87	32 R/W	0x0000 0000
0xFFC0 286C 0xFFC0 2874	MXVR_PAT_EN_0 MXVR_PAT_EN_1	"MXVR Pattern Enable (MXVR_PAT_EN_0, MXVR_PAT_EN_1) Registers" on page 29-88	32 R/W	0x0000 0000
0xFFC0 2878 0xFFC0 287C	MXVR_FRAME_CNT_0 MXVR_FRAME_CNT_1	"MXVR Frame Counter (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1) Registers" on page 29-90	16 R/W	0x0000

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2880 0xFFC0 2884 0xFFC0 2888 0xFFC0 288C 0xFFC0 2890 0xFFC0 2894 0xFFC0 2898 0xFFC0 289C 0xFFC0 28C0 0xFFC0 28A4 0xFFC0 28A8 0xFFC0 28AC 0xFFC0 28B0 0xFFC0 28B4 0xFFC0 28B8	MXVR_ROUTING_0 MXVR_ROUTING_1 MXVR_ROUTING_2 MXVR_ROUTING_3 MXVR_ROUTING_4 MXVR_ROUTING_5 MXVR_ROUTING_6 MXVR_ROUTING_7 MXVR_ROUTING_8 MXVR_ROUTING_9 MXVR_ROUTING_10 MXVR_ROUTING_11 MXVR_ROUTING_12 MXVR_ROUTING_13 MXVR_ROUTING_14	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91	32 WO	0xXXXX XXXX
0xFFC0 28BC	Reserved	–	–	–
0xFFC0 28C0	MXVR_BLOCK_CNT	"MXVR Block Counter (MXVR_BLOCK_CNT) Register" on page 29-94	16 R/W	0x0000
0xFFC0 28C4 to 0xFFC0 28CC	Reserved	–	–	–
0xFFC0 28D0	MXVR_CLK_CTL	"MXVR Clock Control (MXVR_CLK_CTL) Register" on page 29-95	32 R/W	0x0202 0003
0xFFC0 28D4	MXVR_CDRPLL_CTL	"MXVR Clock/Data Recovery PLL Control (MXVR_CDRPLL_CTL) Register" on page 29-101	32 R/W	0x0502 0820

Table 29-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 28D8	MXVR_FMPLL_CTL	"MXVR Frequency Multiply PLL Control (MXVR_FMPLL_CTL) Register" on page 29-104	32 R/W	0x1900 1020
0xFFC0 28DC	MXVR_PIN_CTL	"MXVR Pin Control (MXVR_PIN_CTL) Register" on page 29-106	16 R/W	0x0000
0xFFC0 28E0	MXVR_SCLK_CNT	"MXVR System Clock Counter (MXVR_SCLK_CNT) Register" on page 29-107	16 R/W	0x0000
0xFFC0 28E4 to 0xFFC0 28FF	Reserved	–	–	–

MXVR Configuration (MXVR_CONFIG) Register

The MXVR_CONFIG register sets the configuration of the MXVR node.

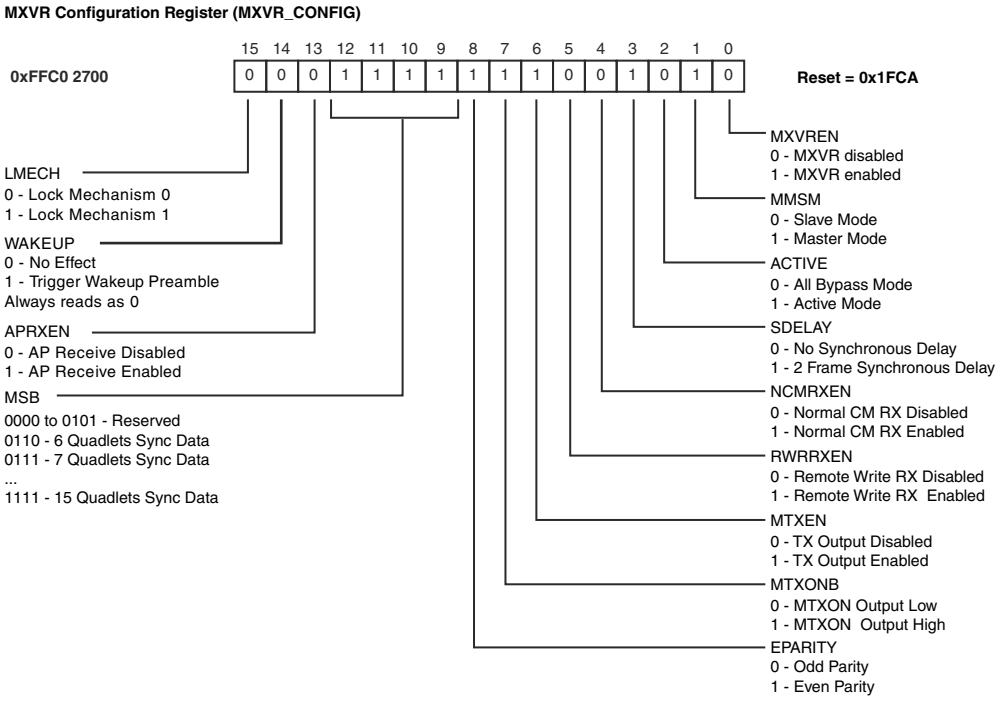


Figure 29-1. MXVR Configuration Register

The MXVR enable (MXVREN) bit enables or disables the MXVR. When the MXVREN bit is set to 0, the MXVR is disabled and is effectively held in a reset state. Disabling the MXVR resets all the MXVR state machines, resets all status bits, and causes the MXVR to enter all bypass-MXVR disabled mode. When the MXVR is in all bypass - MXVR disabled mode the MRX input pin is directly connected to MTX output pin and the MXVR cannot receive or transmit data. When the MXVREN bit is set to 1, the MXVR is enabled and operates based on how the node is configured in the MXVR MMR registers. Note that the MXVR should never be enabled without

the MXVR PLLs enabled and at frequency. Also note that all synchronous data DMA channels should be disabled (by setting MDMAENx = 0) and software should wait until all synchronous data DMA channels are inactive (DMAACTIVEx == 0 and DMAPMENx == 0) before disabling the MXVR.

The MXVR Master Mode/Slave Mode Select (MMSM) bit determines whether the MXVR is the network timing master or is a network slave node. If the MMSM bit is set to 1, the MXVR will be in Master Mode. When in Master Mode, the transmit clock is supplied by the MXVR FMPLL. The transmit clock is then used to generate the data stream transmitted on the MTX pin. In addition, MXVR CDRPLL recovers the receive clock from the incoming data stream received on the MRX pin. The receive clock is then used by the MXVR to sample the incoming data stream.

If the MMSM bit is set to 0, the MXVR will be in Slave Mode. When in Slave Mode, MXVR CDRPLL recovers the receive clock from the incoming data stream received on the MRX pin. The receive clock is then used by the MXVR to sample the incoming data stream. In addition, the receive clock is used to generate the data stream transmitted on the MTX pin.

The Active Mode (ACTIVE) bit determines whether the MXVR will operate in Active Mode or in All Bypass - MXVR Enabled Mode once the MXVR is enabled. When in Active Mode the MXVR can transmit and receive data. When in All Bypass - MXVR Enabled Mode, the MRX input pin is directly connected to MTX and the MXVR can only receive data. When the MXVREN bit is set to 1 and the ACTIVE bit is set to 1, the MXVR will operate in Active Mode. When the MXVREN bit is set to 1 and the ACTIVE bit is set to 0, the MXVR will operate in All Bypass - MXVR Enabled Mode. When the MXVREN bit is set to 0, the ACTIVE bit has no meaning.

The Synchronous Data Delay (SDELAY) bit determines whether the synchronous data field will be delayed by two frames (SDELAY=1) or zero frames (SDELAY=0) passing through the MXVR. In the zero frame delay case the synchronous data will only be delayed by a few bit periods passing through the MXVR. If the MXVR is in All Bypass - MXVR Disabled Mode (MXVREN = 0) or in All Bypass - MXVR Enabled Mode (MXVREN = 1

and ACTIVE = 0), the SDELAY bit has no meaning. If the MXVR is in Active Mode (MXVREN = 1 and ACTIVE = 1) and Master Mode (MMSM = 1), there will always be two frame delays for synchronous data and the SDELAY bit will always read as 1. If the MXVR is in Active Mode (MXVREN = 1 and ACTIVE = 1), Slave Mode (MMSM = 0), and the SDELAY bit is set to 1, there will be two frame delays for synchronous data. If the MXVR is in Active Mode (MXVREN = 1 and ACTIVE = 1), Slave Mode (MMSM = 0), and the SDELAY bit is set to 0, there will only be a few bit delays for synchronous data. Note that synchronous data routing can only be done if the MXVR is the Master and is in Active Mode or if the MXVR is a Slave and is in Active Mode with the SDELAY bit is set to 1.

Table 29-4 lists all possible operating modes of the MXVR, the bit encodings to select the modes, and the functionality in the modes.

Table 29-4. MXVR Operating Modes

Mode	MXVREN	MMSM	ACTIVE	SDELAY	Functionality
All Bypass - MXVR Disabled	0	X	X	X	-Bypassed from MRX to MTX -Cannot receive or transmit SD, AP, and CM -Cannot route or mute SD
Slave Node All Bypass - MXVR Enabled	1	0	0	X	-Bypassed from MRX to MTX -Can only receive SD, AP, and CM -Cannot route or mute SD
Slave Node Active Mode - Zero Frame Delay	1	0	1	0	-SD delayed by few bit periods -Can receive and transmit SD, AP, and CM -Cannot route SD, Can mute SD
Slave Node Active Mode - Two Frame Delay	1	0	1	1	-SD delayed by 2 frame periods -Can receive and transmit SD, AP, and CM -Can route and mute SD
Master Node Active Mode - Two Frame Delay	1	1	1	1	-SD delayed by 2 frame periods -Can receive and transmit SD, AP, and CM -Can route and mute SD
SD = Synchronous Data, AP = Asynchronous Packets, CM = Control Messages					

The Normal Control Message Receive Enable (NCMRXEN) bit determines whether the MXVR is enabled to receive Normal control messages. If the MXVR receives a Normal control message and the NCMRXEN bit is set to 1, the MXVR will write the received data into the Control Message Receive Buffer. If the MXVR receives a Normal control message when the NCMRXEN bit is set to 0, the MXVR will not respond to the Normal control message and will not write to the Control Message Receive Buffer. The NCMRXEN bit is reset to 0.

The Remote Write Receive Enable (RWRRXEN) bit determines whether the MXVR is enabled to receive Remote Write control messages. If the MXVR receives a Remote Write control message and the RWRRXEN bit is set to 1, the MXVR will write the received data into the Remote Read Buffer and will store the MAP and Length values. If the MXVR receives a Remote Write control message when the RWRRXEN bit is set to 0, the MXVR will not write the received data to the Remote Read Buffer and will not write the MAP or Length value. In addition the MXVR will respond to the Remote Write control message with Transmission Status of 0x11 (Not Supported). The RWRRXEN bit is reset to 0.

The MXVR Transmit Data Enable (MTXEN) bit enables or disables the data stream transmitted on the MTX pin when the MXVR is enabled. If the MXVREN bit is set to 1 and the MTXEN bit is set to 0, the MTX pin will remain at a logic low level. If the MXVREN bit is set to 1 and the MTXEN bit is set to 1, the MTX pin will output the transmitted data stream. If the MXVREN bit is set to 0, the MXVR will be in All Bypass - MXVR Disabled Mode and the MTXEN bit has no meaning.

The MXVR Transmit PHY (MTXONB) bit sets the state of the $\overline{\text{MTXON}}$ output pin. The $\overline{\text{MTXON}}$ output pin can be used to gate on and off the power supplied to the Transmit PHY (in the case of MOST®, the Transmit FOT). The $\overline{\text{MTXON}}$ pin can either be operated as a 3.3V compliant output or as an open drain output depending on the state of the MTXONBOD bit in the MXVR_PIN_CTL register. If the MTXONB bit is set to a 0, the $\overline{\text{MTXON}}$ output pin will be driven to a 0V logic-low level. If the MTXONB bit is set to a 1 and the

MTXONBOD bit is set to a 0, the $\overline{\text{MTXON}}$ output pin will be driven to a 3.3V logic-high level. If the MTXONB bit is set to a 1 and the MTXONBOD bit is set to a 1, the $\overline{\text{MTXON}}$ output pin will be three-stated (allowing a pull-up resistor to pull the $\overline{\text{MTXON}}$ output pin to a 5V logic-high level). The MTXONB bit is reset to 1.

The MXVR Even Parity Select (EPARITY) bit indicates whether the parity bit in the frame should be generated with Even Parity or Odd Parity. If the EPARITY bit is set to 0, Odd Parity will be selected. If the EPARITY bit is set to 1, Even Parity will be selected. For MOST®, Even Parity should always be selected. The EPARITY bit is reset to 1.

The synchronous boundary value transmitted by the Master node in a network determines how many quadlets in the frame are dedicated to synchronous data and how many quadlets are dedicated to asynchronous packet data. A quadlet is 4 bytes of data or 4 physical channels in the frame. There are a total of 15 quadlets for synchronous and asynchronous data in the frame. If the synchronous boundary is 6, then 24 bytes will be dedicated to synchronous data and 36 bytes will be dedicated to asynchronous packet data. The MXVR is capable of operating with a synchronous boundary from 0 to 15; however, the MOST® specification limits the synchronous boundary to the range 6 to 14.

When the MXVR is in Master Mode, the value written to the MSB field will be transmitted over the network to all of the slaves as the synchronous boundary. When the MXVR is in Slave Mode, the MSB field is not used. When the MXVR is in either Master Mode or Slave Mode, the synchronous boundary value which was received by the node over the network can be observed as the RSB field in MXVR_STATE_0 register.

The Synchronous Boundary (MSB) field is writable if the MXVR is in Master Mode (MMSM = 1) and is read-only if the MXVR is in Slave Mode (MMSM = 0). Writes to the MSB while in Slave Mode will be ignored and the MSB value will not be effected. Note that a particular procedure must be followed to dynamically change the synchronous boundary for the network to ensure that no data is corrupted and the asynchronous packet channel does not hang.

The Asynchronous Packet Receive Enable (APRXEN) bit determines whether the MXVR is enabled to receive Asynchronous Packets. If the MXVR receives an Asynchronous Packet and the APRXEN bit is set to 1, the MXVR will write the received data into the Asynchronous Packet Receive Buffer. If the MXVR receives an Asynchronous Packet when the APRXEN bit is set to 0, the MXVR will not write the packet to the Asynchronous Packet Receive Buffer. The APRXEN bit is reset to 0.

The Wake-Up (WAKEUP) bit is used to trigger the MXVR when in Master Mode to send the wake-up preamble which will indicate to any node in low-power mode to wake-up. If the MMSM bit is set to 1, writing a 1 to the WAKEUP bit will trigger the MXVR to send the wake-up preamble on the network. If MMSM is set to 0, writing a 1 to the WAKEUP bit will have no effect. Writing a 0 to the WAKEUP bit will have no effect. The WAKEUP bit will always read as 0.

The Lock Mechanism Select (LMECH) bit determines in what order the MXVR Master will send network preambles while locking the network. Lock Mechanism 0 provides the fastest lock time from the completely unlocked state to the super block locked state in a network with only MXVR nodes. Lock Mechanism 1 takes longer than Lock Mechanism 0 to go from the completely unlocked state to the super block locked state; however, if a node in the ring causes an unlock (for example, a node going from All Bypass to Active or vice-versa), only nodes downstream from that node will go unlocked while upstream nodes will remain at their same lock level. Lock Mechanism 1 is generally a better choice for mixed networks which include transceivers other than the MXVR. If the LMECH bit is set to

0, Lock Mechanism 0 is selected. If the LMECH bit is set to 1, Lock Mechanism 1 is selected. When the MXVR is in Slave Mode (MMSM = 0), the LMECH bit has no meaning. The LMECH bit is reset to 0.

MXVR State (MXVR_STATE_0, MXVR_STATE_1) Registers

The MXVR_STATE_x registers indicate the current state of the MXVR. All bits in the MXVR_STATE_x registers are read-only bits.

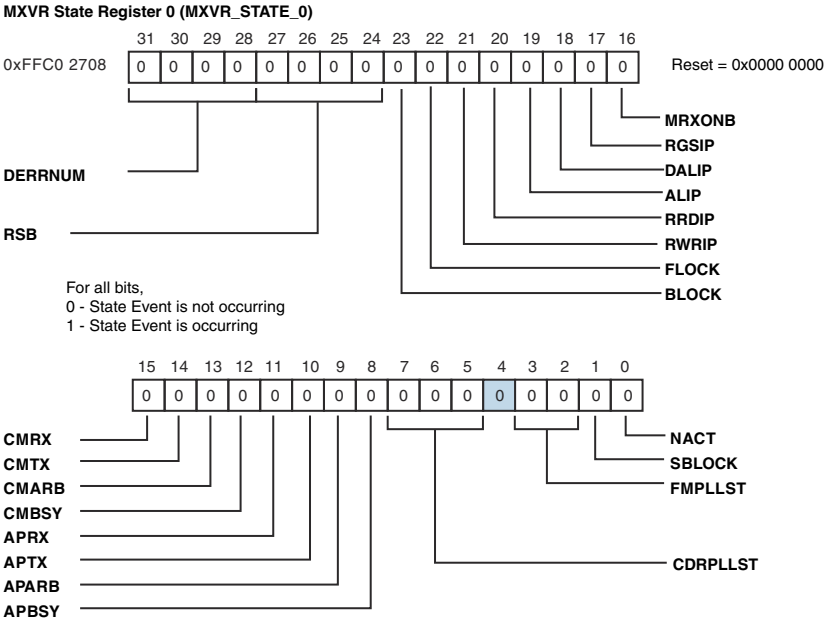


Figure 29-2. MXVR State Register

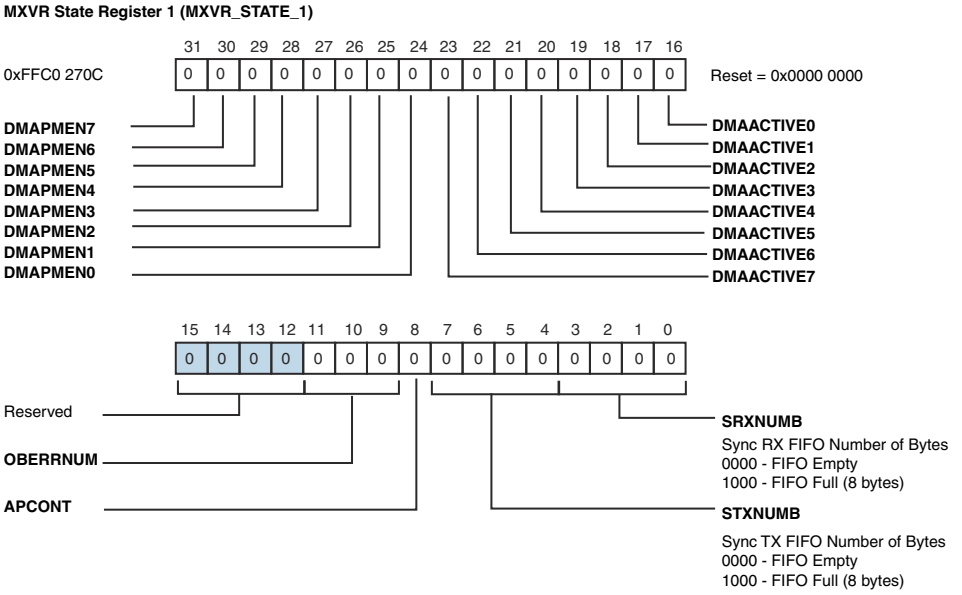


Figure 29-3. MXVR State Register

The Network Active (NACT) bit is a read-only bit which indicates whether the MRX input pin is active. If a single rising edge or falling edge of the MRX pin is detected by the MXVR, the NACT bit will change to a 1. If no rising or falling edges are detected on the MRX input pin for 40 SCLK periods (300 ns for 133 MHz SCLK), the NACT bit will change to a 0. Note that if SCLK is operating at frequency less than 50 MHz, not every edge will be detected.

The Super Block Lock (SBLOCK) bit is a read-only bit which indicates whether the MXVR has locked onto the incoming data stream being received on the MRX input pin and all the preambles are occurring in the right positions. Once the CDRPLL is started-up and the MXVR is enabled, the MXVR will attempt to Super Block Lock. Once the MXVR has Frame Locked, Block Locked, and received the Allocation Table in the correct position in the incoming data stream, the MXVR will be Super Block Locked and the SBLOCK bit will change to 1. If a single preamble is missed or occurs at the wrong position, the MXVR will immediately lose

Super Block Lock and the SBLOCK bit will change to 0. Note that the MXVR can be Super Block Locked when the MXVR is in All Bypass - MXVR Enabled Mode or in Active Mode; however, the MXVR cannot Super Block Lock when the MXVR is in All Bypass - MXVR Disabled Mode.

The MXVR Frequency Multiply PLL State Machine State (FMPLLST) field is a read-only field which gives the current state of the FMPLL State Machine. The FMPLL state encodings are given in Table 29-5.

Table 29-5. FMPLL State Machine States

FMPLLST	FMPLL State Machine State
b#00	FMPLL_RESET
b#01	FMPLL_RAMP
b#10	FMPLL_FMUL
b#11	FMPLL_LOCKED

The MXVR Clock/Data Recovery PLL State Machine State (CDRPLLST) field is a read-only field which gives the current state of the CDRPLL State Machine. The CDRPLL state encodings are given in Table 29-6.

Table 29-6. CDRPLL State Machine States

CDRPLLST	CDRPLL State Machine State
b#000	CDRPLL_RESET
b#001	CDRPLL_RAMP
b#010	CDRPLL_FMUL
b#011	CDRPLL_LOCKED
b#100	CDRPLL_FHOLD
b#101	CDRPLL_ACQUIRE
b#110 - b#111	Reserved

The Asynchronous Packet Transmit Buffer Busy (APBSY) bit is a read-only bit that indicates when the Asynchronous Packet Transmit Buffer is busy transmitting an Asynchronous Packet. The APBSY bit will change to 1 when the STARTAP bit in the MXVR_AP_CTL register is written to 1 which starts the transmission of the packet in the Asynchronous Packet Transmit Buffer. The APBSY bit will change to 0 once the Asynchronous Packet is transmitted or once the Asynchronous Packet being transmitted is successfully cancelled. Note that the Asynchronous Packet Transmit Buffer should never be modified when the APBSY bit is a 1.

The Asynchronous Packet Arbitrating (APARB) bit is a read-only bit that indicates when the MXVR is arbitrating for the asynchronous packet channel so that an asynchronous packet can be transmitted. If the APARB bit is a 1, the MXVR is arbitrating for the asynchronous packet channel. If the APARB bit is a 0, the MXVR is not arbitrating for the asynchronous packet channel. While the MXVR is arbitrating for the asynchronous packet channel, the current asynchronous packet transmission can be cancelled. However, once the MXVR has won arbitration and the asynchronous packet is being transmitted, the transmission cannot be cancelled. Note that when an attempt is made to cancel an asynchronous packet, due to delays in clock synchronization and delays in reading and writing MMRs, the APTS and APTC bits in the MXVR_INT_STAT_1 register should be used to verify whether or not the asynchronous packet was successfully cancelled.

The Asynchronous Packet Transmitting (APTX) bit is a read-only bit that indicates when the MXVR is actively transmitting an asynchronous packet. If the APTX bit is a 1, the MXVR has won arbitration and is in the process of transmitting an asynchronous packet. If the APTX bit is a 0, the MXVR is not in the process of transmitting an asynchronous packet. Once the MXVR has started transmitting an asynchronous packet, the current asynchronous packet transmission cannot be cancelled.

The Asynchronous Packet Receiving (APRX) bit is a read-only bit that indicates when the MXVR is actively receiving an asynchronous packet. If the APRX bit is a 1, the MXVR is in the process of receiving an asynchronous packet. If the APRX bit is a 0, the MXVR is not in the process of receiving an asynchronous packet. Note that the Asynchronous Packet Received (APR) bit in the MXVR_INT_STAT_1 register will change to 1 when the reception of an asynchronous packet has completed and can optionally generate an interrupt.

The Control Message Transmit Buffer Busy (CMBSY) bit is a read-only bit that indicates when the Control Message Transmit Buffer is busy in the process of transmitting a Control Message. The CMBSY bit will change to 1 when the STARTCM bit in the MXVR_CM_CTL register is written to 1 which starts the process of transmitting the Control Message in the Control Message Transmit Buffer. The CMBSY bit will change to a 0 once the Control Message is transmitted or once the Control Message being transmitted is successfully cancelled. Note that the Control Message Transmit Buffer should never be modified when the CMBSY bit is a 1.

The Control Message Arbitrating (CMARB) bit is a read-only bit that indicates when the MXVR is arbitrating for the control message channel so that a control message can be transmitted. If the CMARB bit is a 1, the MXVR is arbitrating for the control message channel. If the CMARB bit is a 0, the MXVR is not arbitrating for the control message channel. While the MXVR is arbitrating for the control message channel, the current control message transmission can be cancelled. However, once the MXVR has won arbitration and the control message is being transmitted, the transmission cannot be cancelled. Note that when an attempt is made to cancel a control message, due to delays in clock synchronization and delays in reading and writing MMRs, the CMTS and CMTC bits in the MXVR_INT_STAT_1 register should be used to verify whether or not the control message was successfully cancelled.

The Control Message Transmitting (CMTX) bit is a read-only bit that indicates when the MXVR is actively transmitting a control message. If the CMTX bit is a 1, the MXVR has won arbitration and is in the process of transmitting a control message. If the CMTX bit is a 0, the MXVR is not in the process of transmitting a control message. Once the MXVR has started transmitting a control message, the current control message transmission cannot be cancelled.

The Receiving Control Message (CMRX) bit is a read-only bit that indicates when the MXVR is actively receiving a Normal control message. If the CMRX bit is a 1, the MXVR is in the process of receiving a Normal control message. If the CMRX bit is a 0, the MXVR is not in the process of receiving a Normal control message. Note that the Control Message Received (CMR) bit in the MXVR_INT_STAT_0 register will change to 1 when the reception of a Normal control message has successfully completed and can optionally generate an interrupt.

The $\overline{\text{MRXON}}$ Input Pin State (MRXONB) bit is a read-only bit which gives the current state of the $\overline{\text{MRXON}}$ input pin. The $\overline{\text{MRXON}}$ input pin should be connected to the optical or electrical PHY status output which indicates whether the PHY is currently receiving data. If the PHY is receiving data, the $\overline{\text{MRXON}}$ input pin will be driven to 0, If the PHY is not receiving data, the $\overline{\text{MRXON}}$ input pin will be driven to 1. A transition from 0 to 1 on the $\overline{\text{MRXON}}$ input pin causes the assertion of the ML2H interrupt event and a transition from 1 to 0 causes the assertion of the MH2L interrupt event. A transition from 1 to 0 on the $\overline{\text{MRXON}}$ input pin can also be used to wake the ADSP-BF54x processor from hibernate state.

The Remote GetSource In Progress (RGSIP) bit is a read-only bit which indicates whether a Remote GetSource system control message is being received and processed by the MXVR.

The Resource De-Allocate In Progress (DALIP) bit is a read-only bit which indicates whether a Resource De-Allocate system control message is being received and processed by the MXVR.

The Resource Allocate In Progress (ALIP) bit is a read-only bit which indicates whether a Resource Allocate system control message is being received and processed by the MXVR.

The Remote Read In Progress (RRDIP) bit is a read-only bit which indicates whether a Remote Read system control message is being received and processed by the MXVR. Note that while a Remote Read is in progress, software should not modify the Remote Read Buffer.

The Remote Write In Progress (RWRIP) bit is a read-only bit which indicates whether a Remote Write system control message is being received and processed by the MXVR. Note that while a Remote Write is in progress, software should not modify the Remote Read Buffer.

The Frame Locked (FLOCK) bit is a read-only bit which indicates whether the MXVR is Frame Locked. Frame Lock is achieved when the CDRPLL has locked onto the received data and the MXVR has detected preambles occurring at the start of every frame. When the FLOCK bit is a 1, the MXVR is Frame Locked and when the FLOCK bit is a 0, the MXVR is not Frame Locked. Once the MXVR Master is Frame Locked and the ring is closed, synchronous data and asynchronous packets can be reliably transmitted and received by all nodes in the ring. Note that the MXVR can be Frame Locked even if the ring network is not closed.

The Block Locked (BLOCK) bit is a read-only bit which indicates whether the MXVR is Block Locked. Block Lock is achieved when the CDRPLL has locked onto the received data, the MXVR has Frame Locked, and the MXVR has received two block preambles in the correct position. When the BLOCK bit is a 1, the MXVR is Block Locked and when the BLOCK bit is a 0, the MXVR is not Block Locked. Once an MXVR node is Block Locked and the ring is closed, control messages can be reliably transmitted and received by all nodes in the ring. Note that the MXVR can be Block Locked even if the ring network is not closed.

The Receive Synchronous Boundary (RSB) field is a read-only field which gives the synchronous boundary value received in the incoming datastream by the MXVR. The RSB value is only valid when the MXVR is Frame Locked.

The DMA Error Channel Number (DERRNUM) field is a read-only field which indicates which DMA Channel caused the last DMA Error (DERR) interrupt event. Table 29-7 gives the DERRNUM encodings and the corresponding DMA channel names. If there are multiple DMA channels causing errors, the DERRNUM will give the value representing the last channel to error prior to the MXVR_STATE_0 register being read.

Table 29-7. DMA Error Number Encodings

DERRNUM	DMA Channel Causing Error
b#0000	Synchronous Data DMA Channel 0
b#0001	Synchronous Data DMA Channel 1
b#0010	Synchronous Data DMA Channel 2
b#0011	Synchronous Data DMA Channel 3
b#0100	Synchronous Data DMA Channel 4
b#0101	Synchronous Data DMA Channel 5
b#0110	Synchronous Data DMA Channel 6
b#0111	Synchronous Data DMA Channel 7
b#1000	Asynchronous Packet Receive DMA Channel
b#1001	Asynchronous Packet Transmit DMA Channel
b#1010	Normal Control Message Receive DMA Channel
b#1011	Control Message Transmit DMA Channel
b#1100	Remote Read Control Message DMA Channel
b#1101	Remote Write Control Message DMA Channel

The Synchronous Receive FIFO Number of Bytes (SRXNUMB) field is a read-only field that indicates how many bytes of data are currently stored in the Synchronous Receive FIFO. The number of bytes can range from 0 (FIFO empty) to 8 (FIFO full).

The Synchronous Transmit FIFO Number of Bytes (STXNUMB) field is a read-only field that indicates how many bytes of data are currently stored in the Synchronous Transmit FIFO. The number of bytes can range from 0 (FIFO empty) to 8 (FIFO full).

The Asynchronous Packet Continuation (APCONT) bit is a read-only bit which indicates the state of the last asynchronous packet continuation bit received over the network. The APCONT bit indicates when the asynchronous packet channel is free and arbitration can occur in the next frame (when APCONT = 0) or when the current asynchronous packet will continue in the next frame (when APCONT = 1).

The DMA Out of Bounds Error Channel Number (OBERRNUM) field is a read-only field which indicates which Synchronous DMA channel caused the last DMA Out of Bounds (OBERR) interrupt event. Table 29-8 gives the OBERRNUM encodings and the corresponding DMA channel names. If there are multiple DMA channels causing errors, the OBERRNUM will give the value representing the last channel to error prior to the MXVR_STATE_0 register being read.

Table 29-8. DMA Out of Bounds Error Number Encodings

OBERRNUM	DMA Channel Causing Error
b#000	Synchronous Data DMA Channel 0
b#001	Synchronous Data DMA Channel 1
b#010	Synchronous Data DMA Channel 2
b#011	Synchronous Data DMA Channel 3
b#100	Synchronous Data DMA Channel 4
b#101	Synchronous Data DMA Channel 5
b#110	Synchronous Data DMA Channel 6
b#111	Synchronous Data DMA Channel 7

The DMAACTIVEx bits indicate whether the DMA channel is active or inactive. When the DMAACTIVEx bit is 1, DMA channel x is active and when the DMAACTIVEx bit is 0, DMA channel x is inactive. Once the MDMAENx bit is set to 1, the exact time when the DMA goes active depends on the Flow Mode selected. When the MDMAENx bit is set to 1 in Stop Mode, the DMA channel will go active on the next frame boundary reached and will stop when the number of programmed transfers is complete. When the MDMAENx bit is set to 1 in Autobuffer Mode, the DMA channel will go active on the next frame boundary and will continue indefinitely. When the MDMAENx bit is set to 1 in Packet-Fixed Count Mode, Packet-Variable Count Mode, or Packet-Start/Stop Mode, the DMA will go active once DMAPMENx is 1 and the “start pattern” is found. When the DMA channel is active in Packet-Fixed Count Mode, the DMA channel will go inactive when the programmed number of transfers is done. When the DMA channel is active in Packet-Fixed Count Mode, the DMA channel will go inactive when the number of transfers specified in the packet are done. When the DMA channel is active in Packet-Start/Stop Mode, the DMA channel will go inactive when the “stop pattern” is found (Packet-Start/Stop). In any flow mode if the MDMAENx bit is set to 0, the DMA channel will go inactive and disable on the next frame boundary reached.

The DMAPMENx bits indicate whether the DMA channel is enabled for Pattern Matching. In Packet-Fixed Count Mode, Packet-Variable Count Mode, or Packet-Start/Stop Mode, when the MDMAENx bit is set to 1, the DMA channel will be enabled for pattern matching on the next frame boundary reached. The DMA channel will remain enabled for pattern matching until the MDMAENx bit is set to 0. Once the MDMAENx bit is set to 0, the DMA channel will be disabled on the next frame boundary reached.

MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)

The MXVR_INT_STAT_0 register indicates the current status of all events that can generate a Status Change Interrupt or a Control Message Interrupt in the MXVR. Each bit in the MXVR_INT_STAT_0 indicates whether a particular event has occurred. If the corresponding interrupt enable bit in the MXVR_INT_EN_0 is set to 1, the occurrence of that event will generate an interrupt.

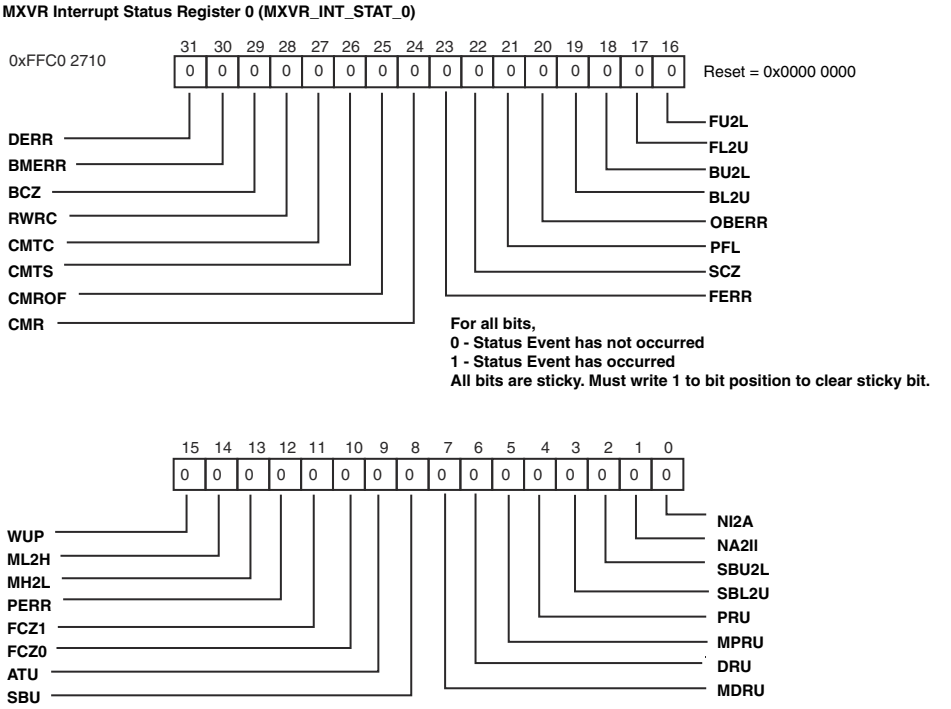


Figure 29-4. MXVR Interrupt Status Register 0

The following status events generate the Status Change Interrupt: NI2A, NA2I, SBU2L, SBL2U, PRU, MPRU, DRU, MDRU, SBU, ATU, FCZ0, FCZ1, PERR, MH2L, ML2H, WUP, FU2L, FL2U, BU2L, BL2U, OBERR, PFL, SCZ, FERR, BCZ, BMERR and DERR.

The following status events generate the Control Message Interrupt: CMR, CMROF, CMTS, and CMTC, and RWRC.

All bits in the MXVR_INT_STAT_0 register are sticky bits. The sticky bits are set to 1 when an event occurs, but must be written with a 1 in order to clear the bit.

The Network Inactive to Active (NI2A) interrupt event will change to 1 when the Network Activity (NACT) bit changes from Inactive (NACT = 0) to Active (NACT = 1). If the NI2AEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of NI2A will generate a Status Change Interrupt. The NI2A bit can be cleared by writing a 1 to the NI2A bit position.

The Network Active to Inactive (NA2I) interrupt event will change to 1 when the Network Activity State (NACT) bit changes from Active (NACT = 1) to Inactive (NACT = 0). If the NA2IEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of NA2I will generate a Status Change Interrupt. The NA2I bit can be cleared by writing a 1 to the NA2I bit position.

The Super Block Unlocked to Locked (SBU2L) interrupt event will change to 1 when the Super Block Locked State (SBLOCK) bit changes from Super Block Unlocked (SBLOCK = 0) to Super Block Locked (SBLOCK = 1). If the SBU2LEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of SBU2L will generate a Status Change Interrupt. The SBU2L bit can be cleared by writing a 1 to the SBU2L bit position.

The Super Block Locked to Unlocked (SBL2U) interrupt event will change to 1 when the Super Block Locked State (SBLOCK) bit changes from Super Block Locked (SBLOCK = 1) to Super Block Unlocked (SBLOCK = 0). If the SBL2UEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of SBL2U will generate a Status Change Interrupt. The SBL2U bit can be cleared by writing a 1 to the SBL2U bit position.

The Position Register Updated (PRU) interrupt event will change to 1 when the node position becomes valid after lock or whenever the node position changes once valid. PRU will assert when the PVALID bit in the MXVR_POSITION register changes from 0 to 1 or when the POSITION field in the MXVR_POSITION register changes while the PVALID bit is a 1. If the PRUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of PRU will generate a Status Change Interrupt. The PRU bit can be cleared by writing a 1 to the PRU bit position. Note that the PRU interrupt event will never

occur when the MXVR is enabled in Master Mode. In Master Mode, the PVALID bit will be set to 1 immediately after the MXVR is enabled, but no PRU interrupt event will be generated.

The Maximum Position Register Updated (MPRU) interrupt event will change to 1 when the maximum position becomes valid after lock or whenever the maximum position changes once valid. MPRU will assert when the MPVALID bit in the MXVR_MAX_POSITION register changes from 0 to 1 or when the MPOSITION field in the MXVR_MAX_POSITION register changes while the MPVALID bit is a 1. If the MPRUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of MPRU will generate a Status Change Interrupt. The MPRU bit can be cleared by writing a 1 to the MPRU bit position.

The Delay Register Updated (DRU) interrupt event will change to 1 when the delay becomes valid after lock or whenever the delay changes once valid. DRU will assert when the DVALID bit in the MXVR_DELAY register changes from 0 to 1 or when the DELAY field in the MXVR_DELAY register changes while the DVALID bit is a 1. If the DRUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of DRU will generate a Status Change Interrupt. The DRU bit can be cleared by writing a 1 to the DRU bit position. Note that the DRU interrupt event will never occur when the MXVR is enabled in Master Mode. In Master Mode, the DVALID bit will be set to 1 immediately after the MXVR is enabled, but no DRU interrupt event will be generated.

The Maximum Delay Register Updated (MDRU) interrupt event will change to 1 when the maximum delay becomes valid after lock or when the maximum delay changes once valid. MDRU will assert when the MDVALID bit in the MXVR_MAX_DELAY register changes from 0 to 1 or when the MDELAY field in the MXVR_MAX_DELAY register changes while the MDVALID bit is a 1. If the MDRUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of MDRU will generate a Status Change Interrupt. The MDRU bit can be cleared by writing a 1 to the MDRU bit position.

The Synchronous Boundary Updated (SBU) interrupt event will change to 1 when the MXVR is Frame Locked and the Synchronous Boundary information received over the network changes. When the Synchronous Boundary information received over the network changes, the Received Synchronous Boundary (RSB) field in the MXVR_STATE_0 register will be updated. The SBU bit will only change to 1 in a node operating in Slave Mode (MMSM = 0). If the SBUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of SBU will generate a Status Change Interrupt. The SBU bit can be cleared by writing a 1 to the SBU bit position.

The Allocation Table Updated (ATU) interrupt event indicates when the allocation table is updated. When the MXVR is in Master Mode (MMSM = 1), ATU will change to 1 whenever a Resource Allocate or a Resource De-Allocate control message is received and processed or when the Allocation Table is received over the network (once every 1024 frames). When in Slave Mode (MMSM = 0), ATU will assert when the Allocation Table is received over the network (once every 1024 frames). The MXVR does not determine whether the Allocation Table has changed—only that the Allocation Table is received. Software must read the Allocation Table registers to determine if any changes have been made. If the ATUEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of ATU will generate a Status Change Interrupt. The ATU bit can be cleared by writing a 1 to the ATU bit position. Note that it is recommended to only read the Allocation Table (MXVR_ALLOC_x registers) either immediately following an ATU event or immediately following a BCZ event to avoid the possibility of reading the Allocation Table while it is in the process of being updated.

The Parity Error (PERR) interrupt event will change to 1 whenever the calculated parity of the received frame does not match the parity bit in that frame. If the PERREN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of PERR will generate an Status Change Interrupt. The PERR bit can be cleared by writing a 1 to the PERR bit position.

The MRXONB Low to High (ML2H) interrupt event will change to 1 when the MRXONB bit in the MXVR_STATE_0 register changes from low to high, indicating that the $\overline{\text{MRXON}}$ input pin has changed from low to high (“light on” to “light off”). If the ML2HEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of ML2H will generate a Status Change Interrupt. The ML2H bit can be cleared by writing a 1 to the ML2H bit position.

The MRXONB High to Low (MH2L) interrupt event will change to 1 when the MRXONB bit in the MXVR_STATE_0 register changes from high to low, indicating that the $\overline{\text{MRXON}}$ input pin has changed from high to low (“light off” to “light on”). If the MH2LEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of MH2L will generate a Status Change Interrupt. The MH2L bit can be cleared by writing a 1 to the MH2L bit position.

The Wake-Up Preamble Received (WUP) interrupt event will change to 1 when a Wake-Up Preamble is received over the network by the MXVR. The WUP bit will assert regardless of the current operating mode of the MXVR. If the WUPEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of WUP will generate a Status Change Interrupt. The WUP bit can be cleared by writing a 1 to the WUP bit position.

The Frame Counter 0 Zero (FCZ0) interrupt event will change to 1 when Frame Counter 0 is started by writing a value to the MXVR_FRAME_CNT_0 register and Frame Counter 0 has decremented down to zero. If the FCZ0EN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of FCZ0 will generate an Status Change Interrupt. The FCZ0 bit can be cleared by writing a 1 to the FCZ0 bit position.

The Frame Counter 1 Zero (FCZ1) interrupt event will change to 1 when Frame Counter 1 is started by writing a value to the MXVR_FRAME_CNT_1 register and Frame Counter 1 has decremented down to zero. If the FCZ1EN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of FCZ1 will generate an Status Change Interrupt. The FCZ1 bit can be cleared by writing a 1 to the FCZ1 bit position.

The Frame Unlocked to Locked (FU2L) interrupt event will change to 1 when the Frame Locked (FLOCK) bit in the MXVR_STATE_0 register changes from Frame Unlocked (FLOCK=0) to Frame Locked (FLOCK=0). If the FU2LEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of the FU2L will generate a Status Change Interrupt. The FU2L bit can be cleared by writing a 1 to the FU2L bit position.

The Frame Locked to Unlocked (FL2U) interrupt event will change to 1 when the Frame Locked (FLOCK) bit in the MXVR_STATE_0 register changes from Frame Locked (FLOCK=1) to Frame Unlocked (FLOCK=0). If the FL2UEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of the FL2U will generate a Status Change Interrupt. The FL2U bit can be cleared by writing a 1 to the FL2U bit position.

The Block Unlocked to Locked (BU2L) interrupt event will change to 1 when the Block Locked (BLOCK) bit in the MXVR_STATE_0 register changes from Block Unlocked (BLOCK=0) to Block Locked (BLOCK=0). If the BU2LEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of the BU2L will generate a Status Change Interrupt. The BU2L bit can be cleared by writing a 1 to the BU2L bit position.

The Block Locked to Unlocked (BL2U) interrupt event will change to 1 when the Block Locked (BLOCK) bit in the MXVR_STATE_0 register changes from Block Locked (BLOCK=1) to Block Unlocked (BLOCK=0). If the BL2UEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of the BL2U will generate a Status Change Interrupt. The BL2U bit can be cleared by writing a 1 to the BL2U bit position.

The DMA Out of Bounds Error (OBERR) interrupt event indicates when a Synchronous DMA channel attempts to move outside its allocated memory buffer. A DMA Out of Bounds Error can only occur for channels operating in the Synchronous Packet-Variable Count mode or the Synchronous Packet-Stat/Stop mode. The allocated memory buffer is defined by the values programmed in the MXVR_DMAx_START_ADDR register and the MXVR_DMAx_COUNT register. A DMA Out of Bounds Error is either a result of a bit error occurring in data being received (for example, a bit error

causing the variable count value to be received incorrectly or a bit error causing the stop pattern to be missed) or a result of the synchronous packet being transmitted incorrectly (for example, the transmitted synchronous packet being larger than the allocated memory buffer). When a DMA Out of Bounds Error is detected, the DMA channel is automatically disabled, the OBERR bit is set to 1, and the OBERRNUM field in the MXVR_STATE_0 register indicates the channel which generated the error. If the OBERREN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of OBERR will generate an Status Change Interrupt. The OBERR bit can be cleared by writing a 1 to the OBERR bit position.

The PLL Frequency Locked (PFL) interrupt event indicates when the FMPLL or the CDRPLL transition to their frequency locked states. When the FMPLL State Machine State (FMPLLST) transitions from the FMPLL_FMUL state to the FMPLL_LOCKED state or the CDRPLL State Machine State (CDRPLLST) transitions from the CDRPLL_FMUL state to the CDRPLL_FHOLD state, the PFL bit will be set to 1. The FMPLLST and the CDRPLLST can be read in the MXVR_STATE_0 register. If the PFLEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of PFL will generate an Status Change Interrupt. The PFL bit can be cleared by writing a 1 to the PFL bit position.

The System Clock Counter Zero (SCZ) interrupt event will change to 1 when the System Clock Counter is started by writing a value to the MXVR_SCLK_CNT register and System Clock Counter has decremented down to zero. If the SCZEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of SCZ will generate an Status Change Interrupt. The SCZ bit can be cleared by writing a 1 to the SCZ bit position.

The FIFO Error (FERR) interrupt event will change to 1 when one of the MXVR internal FIFO's overflows or underflows. This condition will most likely cause data corruption. This is a catastrophic event and the MXVR will automatically disable the effected transmit DMA channels. The internal FIFO underflows and overflows occur when the MXVR DMA channels cannot get enough internal DMA bus bandwidth for transfers to

and from L1 to support the network interface. This normally would only happen if the system clock and/or the core clock frequency are lowered to a point where the internal busses cannot provide enough bandwidth to support all the enabled peripherals. If the FERREN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of FERR will generate a Status Change Interrupt. The FERR bit can be cleared by writing a 1 to the FERR bit position. If the FERR occurs due to the Synchronous Transmit FIFO underflowing all Synchronous Data Transmit DMA channels will automatically be disabled. If the FERR occurs due to the Asynchronous Packet Transmit FIFO underflowing, the Asynchronous Packet Transmit DMA channel will be disabled. If the FERR occurs due to the Synchronous Receive FIFO or the Asynchronous Packet Receive FIFO overflowing, the associated DMA channels will not automatically be disabled; however, the data received should be assumed to be corrupted.

If the FERR event ever occurs when running an application, the application code should be changed (the system clock and/or core clock frequency should be increased or the amount of DMA bandwidth being used should be decreased). The FERR event should never be allowed to occur in an application as this event indicates that data corruption may be occurring. In addition, if the FERR event occurs, the MXVR must be disabled and re-enabled in order to reset the internal FIFOs prior to re-enabling DMA channels.

The Control Message Received (CMR) interrupt event will change to 1 once a complete control message is received by the MXVR and stored into the Control Message Receive Buffer. The CMR bit will not be set for System control messages or Normal control messages that fail the CRC check. If the CMREN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of CMR will generate a Control Message Interrupt. The CMR bit can be cleared by writing a 1 to the CMR bit position.

The Control Message Receive Buffer Overflow (CMROF) interrupt event will change to 1 when the Control Message Receive Buffer is full and a new control message is received over the network by the MXVR. The control message that is received by the MXVR when the Control Message Receive Buffer is full will be completely lost (the MXVR will respond with "Buffer Full" transmission status). If the CMROFEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of CMROF will generate a Control Message Interrupt. The CMROF bit can be cleared by writing a 1 to the CMROF bit position.

The Control Message Transmit Buffer Successfully Sent (CMTS) interrupt event will change to 1 when the complete control message in the Control Message Transmit Buffer is transmitted and the Transmission Status received back after the message has circled the network is updated in the Control Message Transmit Buffer. If the CMTSEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of CMTS will generate a Control Message Interrupt. The CMTS bit can be cleared by writing a 1 to the CMTS bit position.

The Control Message Transmit Buffer Successfully Cancelled (CMTC) interrupt event will change to 1 when the transmission of the control message in the Control Message Transmit Buffer is cancelled. The transmission of the control message can only be cancelled while the MXVR is arbitrating for the control message channel. Once the MXVR has won arbitration, the transmission cannot be cancelled. If the CMTCEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of CMTC will generate a Control Message Interrupt. The CMTC bit can be cleared by writing a 1 to the CMTC bit position.

The Remote Write Control Message Complete (RWRC) interrupt event will change to 1 when an incoming Remote Write Control Message is processed and the received data is DMA'ed to the Remote Read Buffer and the received write address and write length have also been DMA'ed to the Remote Read Buffer. If the RWRCEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of RWRC will generate a Control Message Interrupt.

The RWRC bit can be cleared by writing a 1 to the RWRC bit position. The RWRC bit used by software to know when the Remote Read Buffer is written to by another node.

The Block Counter Zero (BCZ) interrupt event will change to 1 when the Block Counter is started by writing a value to the MXVR_BLOCK_CNT register and Block Counter has decremented down to zero. If the BCZEN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of BCZ will generate an Status Change Interrupt. The BCZ bit can be cleared by writing a 1 to the BCZ bit position. Note that the Block Counter only decrements at the beginning of Normal Blocks and not on the blocks containing the Allocation Table.

The Biphase Mark Coding Error (BMERR) interrupt event will change to 1 when there is a biphase mark code violation in any part of the frame other than the expected code violations in the preambles. If the BMERREN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of BMERR will generate a Status Change Interrupt. The BMERR bit can be cleared by writing a 1 to the BMERR bit position.

The DMA Error (DERR) interrupt event will change to 1 when one of the DMA channels encounters an error. DMA errors occur when the DMA channel attempts to access an illegal address in L1 or L2 memory. The DMA Error Number (DERRNUM) field in the MXVR_STATE_0 register gives a value which indicates which DMA channel was the last to cause a DMA error. When a DMA channel encounters an error, the channel will be disabled automatically at the point where the error occurred. If the DERREN bit is set to 1 in the MXVR_INT_EN_0 register, the assertion of DERR will generate a Status Change Interrupt. The DERR bit can be cleared by writing a 1 to the DERR bit position.

MXVR Interrupt Status_1 (MXVR_INT_STAT_1) Register

The MXVR_INT_STAT_1 register indicates the current status of all events that can generate a Synchronous Data Interrupt or an Asynchronous Packet Interrupt.

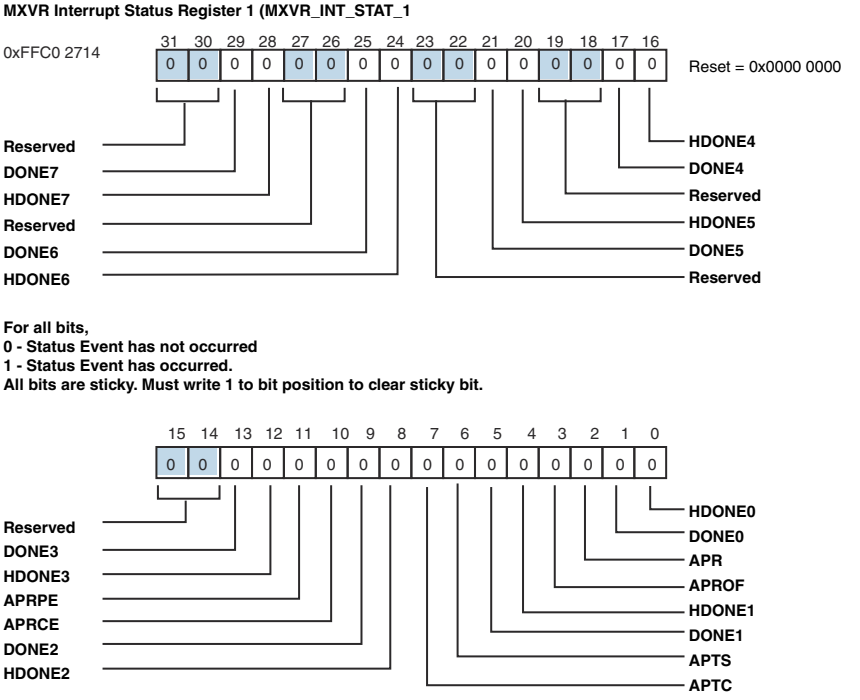


Figure 29-5. MXVR Interrupt Status Register_1

Each bit in the MXVR_INT_STAT_1 indicates whether a particular event has occurred. If the corresponding interrupt enable bit in the MXVR_INT_EN_1 is set to 1, the occurrence of that event will generate an interrupt.

The following status events will generate a Synchronous Data Interrupt: HDONE0, DONE0, HDONE1, DONE1, HDONE2, DONE2, HDONE3, DONE3, HDONE4, DONE4, HDONE5, DONE5, HDONE6, DONE6, HDONE7, and DONE7. The following status events will generate an Asynchronous Packet Interrupt: APR, APROF, APTS, APTC, APRCE, and APRPE.

All bits in the MXVR_INT_STAT_1 register are sticky bits. The sticky bits are set to 1 when an event occurs, but must be written with a 1 in order to clear the bit.

The DMAx Half-Done (HDONEx) interrupt event will change to 1 when DMA channel x has completed half of the programmed transfers for the current block in Stop or Autobuffer Mode or when DMA channel x has completed an odd numbered packet in one of the Synchronous Packet Modes. If the HDONEx bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of HDONEx will generate a Synchronous Data DMA Interrupt. The HDONEx bit can be cleared by writing a 1 to the HDONEx bit position.

The DMAx Done (DONEx) interrupt event will change to 1 when DMA channel x has completed all of the programmed transfers for the current block in Stop or Autobuffer Mode or when DMA channel x has completed an even numbered packet in one of the Synchronous Packet Modes. If the DONEx bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of DONEx will generate a Synchronous Data DMA Interrupt. The DONEx bit can be cleared by writing a 1 to the DONEx bit position.

The Asynchronous Packet Received (APR) interrupt event will change to 1 once a complete asynchronous packet is received by the MXVR and stored into the Asynchronous Packet Receive Buffer. If the APREN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APR will generate an Asynchronous Packet Interrupt. The APR bit can be cleared by writing a 1 to the APR bit position.

The Asynchronous Packet Receive Buffer Overflow (APROF) interrupt event will change to 1 when the Asynchronous Packet Receive Buffer is full and a new asynchronous packet is received over the network by the

MXVR. The asynchronous packet that is received by the MXVR when the Asynchronous Packet Receive Buffer is full will be completely lost. If the APROFEN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APROF will generate an Asynchronous Packet Interrupt. The APROF bit can be cleared by writing a 1 to the APROF bit position.

The Asynchronous Packet Transmit Buffer Successfully Sent (APTS) interrupt event will change to 1 when the complete asynchronous packet in the Asynchronous Packet Transmit Buffer is transmitted. If the APTSEN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APTS will generate an Asynchronous Packet Interrupt. The APTS bit can be cleared by writing a 1 to the APTS bit position.

The Asynchronous Packet Transmit Buffer Successfully Cancelled (APTC) interrupt event will change to 1 when the transmission of the asynchronous packet in the Asynchronous Packet Transmit Buffer is cancelled. The transmission of the asynchronous packet can only be cancelled while the MXVR is arbitrating for the asynchronous packet channel. Once the MXVR has won arbitration, the transmission cannot be cancelled. If the APTCEN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APTC will generate an Asynchronous Packet Interrupt. The APTC bit can be cleared by writing a 1 to the APTC bit position.

The Asynchronous Packet Receive CRC Error (APRCE) interrupt event will change to 1 when an Asynchronous Packet was received with a CRC Error. The Asynchronous Packet that was received by the MXVR with a CRC error will not be stored into the Asynchronous Packet Receive Buffer. If the APRCEEN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APRCE will generate an Asynchronous Packet Interrupt. The APRCE bit can be cleared by writing a 1 to the APRCE bit position.

The Asynchronous Packet Receive Packet Error (APRPE) interrupt event will change to 1 when an Asynchronous Packet was received and the Length stored as part of the Asynchronous Packet did not match the length of the Asynchronous Packet which was actually received or if the Asynchronous Packet Continuation (APCONT) bit gets corrupted. If a

Packet Error is detected, and there is an Asynchronous Packet which is started and is waiting to win arbitration, the MXVR will automatically cancel the transmission and the APTC will be set to 1. In addition, the Asynchronous Packet which was being received when the Packet Error occurred will not be stored in the Asynchronous Packet Receive Buffer. If the APRPEEN bit is set to 1 in the MXVR_INT_EN_1 register, the assertion of APRPE will generate an Asynchronous Packet Interrupt. The APRPE bit can be cleared by writing a 1 to the APRPE bit position. Note that the MXVR Master can resolve packet errors by asserting the RESETAP bit in the MXVR_AP_CTL register.

MXVR Interrupt Enable 0 (MXVR_INT_EN_0) Register

The MXVR_INT_EN_0 register is used to enable or disable the generation of an interrupt when a particular event occurs in MXVR_INT_STAT_0. The interrupt enables in the MXVR_INT_EN_0 register correspond on a bit-to-bit basis with the events in the MXVR_INT_STAT_0 register. If an interrupt enable bit is set to 1, whenever the corresponding event bit in the MXVR_INT_STAT_0 register is asserted, the associated MXVR interrupt will be asserted and whenever the event bit is negated, the associated MXVR interrupt will be negated (assuming no other events are causing that interrupt to be asserted). If the interrupt enable bit is set to 0, the associated interrupt output will not assert when the corresponding event bit asserts.

Note that interrupt outputs remain asserted as long as the event bit and the interrupt enable bit are asserted. For the event bits which are sticky bits, the Interrupt Service Routine must write a 1 to the asserted event bit position in the MXVR_INT_STAT_0 register in order to clear the event bit.

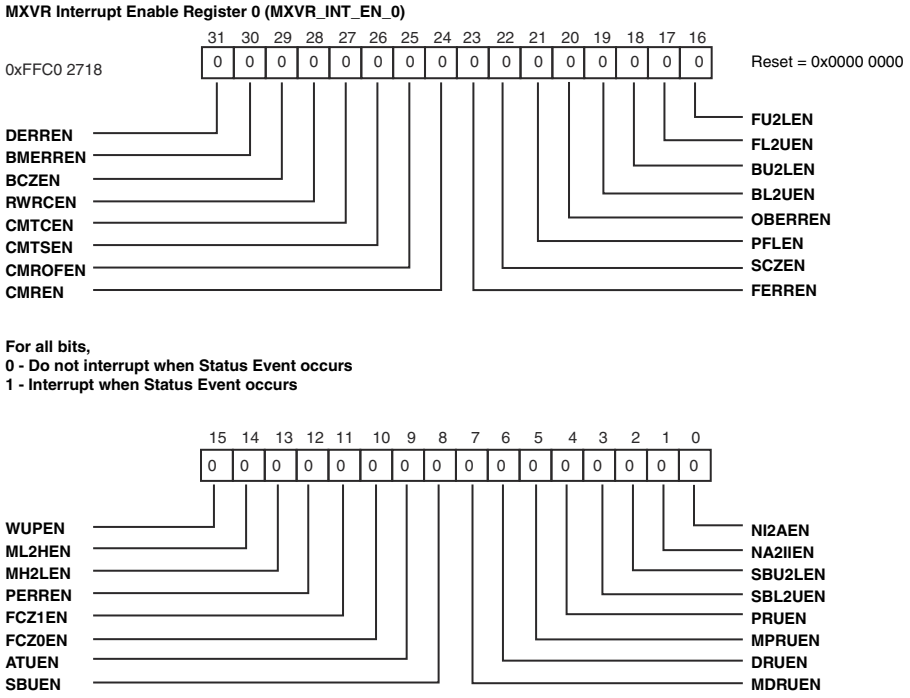


Figure 29-6. MXVR Interrupt Enable Register 0

The MXVR_INT_EN_0 register contains the following interrupt enables:

- Network Inactive to Active interrupt enable (NI2AEN)
- Network Active to Inactive interrupt enable (NA2IEN)
- Super Block Unlocked to Locked interrupt enable (SBU2LEN)
- Super Block Locked to Unlocked interrupt enable (SBL2UEN)
- Position Register Updated interrupt enable (PRUEN)
- Maximum Position Register Updated interrupt enable (MPRUEN)
- Delay Register Updated interrupt enable (DRUEN)

- Maximum Delay Register Updated interrupt enable (MDRUEN)
- Synchronous Boundary Updated interrupt enable (SBUEN)
- Allocation Table Updated interrupt enable (ATUEN)
- Parity Error interrupt enable (PERREN)
- MRXONB High to Low interrupt enable (MH2LEN)
- MRXONB Low to High interrupt enable (ML2HEN)
- Wakeup Preamble Detected interrupt enable (WUPEN)
- Frame Unlocked To Locked interrupt enable (FU2LEN)
- Frame Locked to Unlocked interrupt enable (FU2UEN)
- Block Unlocked to Locked interrupt enable (BU2LEN)
- Block Locked to Unlocked interrupt enable (BL2UEN)
- DMA Out of Bounds Error interrupt enable (OBERREN)
- PLL Frequency Locked interrupt enable (PFLEN)
- System Clock Counter Zero interrupt enable (SCZEN)
- FIFO Error interrupt enable (FERREN)
- Frame Counter 0 Zero interrupt enable (FCZ0EN)
- Frame Counter 1 Zero interrupt enable (FCZ1EN)
- Control Message Received interrupt enable (CMREN)
- Control Message Receive Buffer Overflow interrupt enable
- Control Message Transmit Buffer Successfully Sent interrupt enable (CMTSEN)

- Control Message Transmit Buffer Successfully Cancelled interrupt enable (CMTCEN)
- Remote Write Complete interrupt enable (RWRCEN)
- Block Counter Zero interrupt enable (BCZEN)
- Biphase Mark Coding Error interrupt enable (BMERREN)
- DMA Error interrupt enable (DERREN)

MXVR Interrupt Enable 1 (MXVR_INT_EN_1) Register

The MXVR_INT_EN_1 register is used to enable or disable the generation of an interrupt when a particular event occurs in MXVR_INT_STAT_1. The interrupt enables in the MXVR_INT_EN_1 register correspond on a bit-to-bit basis with the events in the MXVR_INT_STAT_1 register. If an interrupt enable bit is set to 1, whenever the corresponding event bit in the MXVR_INT_STAT_1 register is asserted, the associated MXVR interrupt will be asserted and whenever the event bit is negated, the associated MXVR interrupt will be negated (assuming no other events are causing that interrupt to be asserted). If the interrupt enable bit is set to 0, the associated interrupt output will not assert when the corresponding event bit asserts.

Note that interrupt outputs remain asserted as long as the event bit and the interrupt enable bit are asserted. For the event bits which are sticky bits, the Interrupt Service Routine must write a 1 to the asserted event bit position in the MXVR_INT_STAT_1 register in order to clear the event bit.

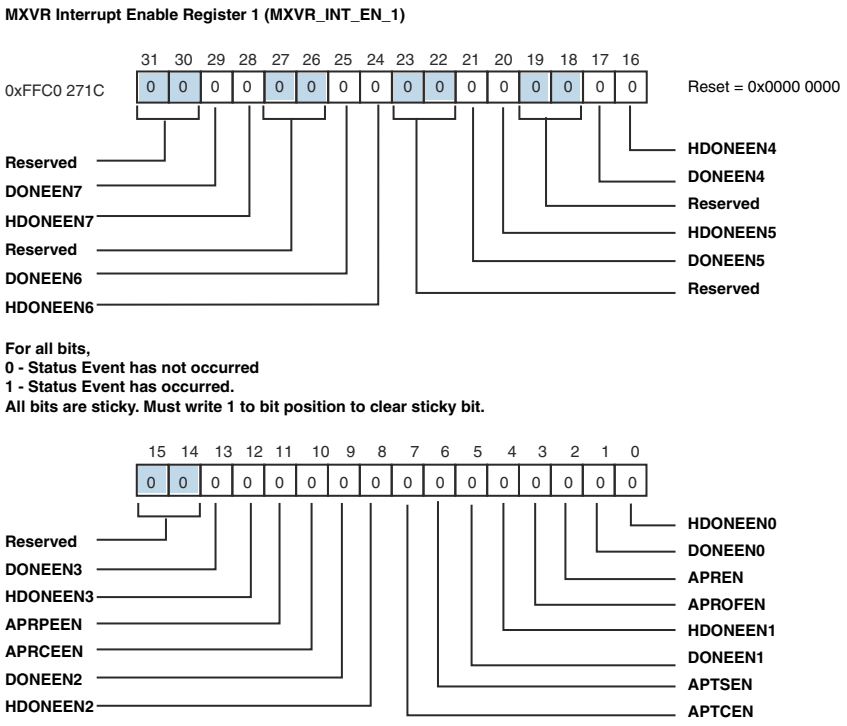


Figure 29-7. MXVR Interrupt Enable Register 1

The MXVR_INT_EN_1 register contains the following interrupt enables:

- DMA Channel x Half Done interrupt enable (HDONEENx)
- DMA Channel x Done interrupt enable (DONEENx)
- Asynchronous Packet Received interrupt enable (APREN)
- Asynchronous Packet Receive Buffer Overflow interrupt enable (APROFEN)
- Asynchronous Packet Transmit Buffer Successfully Sent interrupt enable (APTSEN)

- Asynchronous Packet Transmit Buffer Successfully Cancelled interrupt enable (APTCEN)
- Asynchronous Packet Receive CRC Error interrupt enable (APRCEEN)
- Asynchronous Packet Receive Packet Error interrupt enable (APRPEEN)

MXVR Node Position (MXVR_POSITION) Register

The MXVR_POSITION register is a read-only register that indicates the MXVR's physical node position within the ring network. The Master node is always at position 0. The Slave nodes in the network have their physical positions checked constantly over the network. If the PVALID bit is a 1, then the POSITION field is valid and indicates the MXVR's physical node position. If the PVALID bit is a 0, then the POSITION field is not valid. The physical node position can range from 0 to 63.

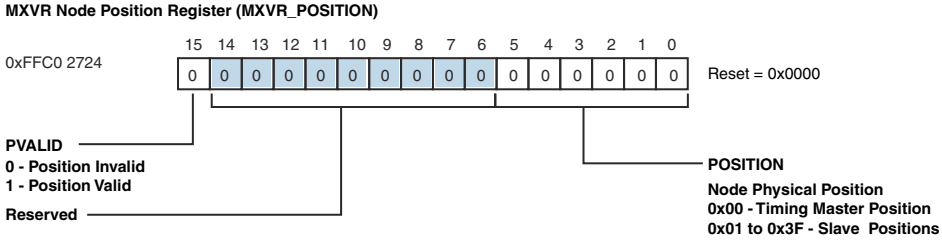


Figure 29-8. MXVR Node Position Register

When the MXVR is disabled, the PVALID bit will be 0. When the MXVR is enabled in Master Mode, the PVALID bit will be 1 and the POSITION field will be 0. Once the MXVR is enabled in Master Mode and the PVALID bit is 1, only asserting reset or disabling the MXVR will cause the PVALID bit to change to 0. When the MXVR is enabled in Slave Mode, the PVALID bit will be 0 until the MXVR has reached a lock level at which the node position can be correctly determined from the incoming datastream. Once the

node position is correctly determined, the PVALID bit will change to a 1 and the POSITION field will contain the physical node position. Subsequent changes to the node position (for example, upstream nodes entering or exiting All Bypass) will cause the POSITION field to update, but the PVALID bit will remain a 1 as long the MXVR remains locked throughout the change. Once the MXVR is enabled in Slave Mode and the PVALID bit is 1, only asserting reset, disabling the MXVR, or losing lock will cause the PVALID bit to change to 0.

MXVR Maximum Node Position (MXVR_MAX_POSITION) Register

The MXVR_MAX_POSITION register is a read-only register that indicates the total number of Active nodes within the ring network. The Slave nodes in the network have the MPOSITION field updated once every 1024 frames. If the MPVALID bit is a 1, then the MPOSITION field is valid. If the MPVALID bit is a 0, then the MPOSITION field is not valid. The maximum physical node position can range from 1 (MPOSITION= b#000001) to 64 (MPOSITION=b#000000).

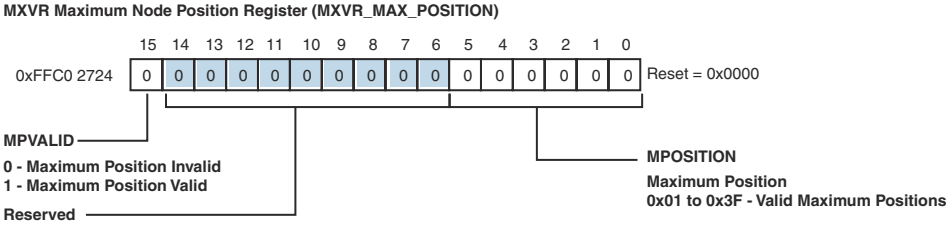


Figure 29-9. MXVR Maximum Node Position Register

Once the Master has achieved a lock level at which the total number of nodes in the network can accurately be determined, the MPOSITION field will be updated, the MPVALID bit will change to a 1 in the Master. At that point the Master will distribute the MPOSITION value to all the Slave nodes ever 1024 frames. Once the Slave nodes have achieved a lock level at

which the MPOSITION value distributed by the Master can be accurately received, the MPOSITION field will be updated and the MPVALID bit will change to a 1 in the Slave nodes. Subsequent changes to the total number of nodes in the network (for example, nodes entering or exiting All Bypass) will cause the MPOSITION field to update, but the MPVALID bit will remain a 1 as long as the MXVR remains locked throughout the change.

Once MPVALID is set to 1, only asserting reset, disabling the MXVR, or losing lock will cause the MPVALID to change to a 0.

MXVR Node Frame Delay (MXVR_DELAY) Register

The MXVR_DELAY register is a read-only register that indicates the number of nodes with 2 frame delays that synchronous data will pass through when going from the transmit output of the Master over the network to the receive input of this MXVR node. The DELAY field value is calculated by determining the number of Slave nodes operating in Active Mode with 2 frame delays between the Master the MXVR node. The DELAY field value is calculated once every 1024 frames.

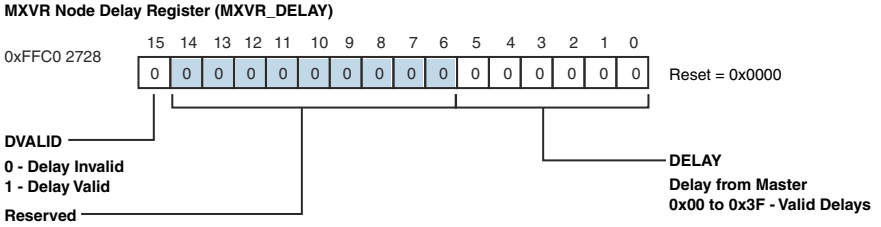


Figure 29-10. MXVR Node Frame Delay Register

If the DVALID bit is a 1, then the DELAY field is valid. If the DVALID bit is a 0, then the DELAY field is not valid. The DELAY field can range from 0 to 63 (representing from 0 to 126 frame delays for synchronous data).

When the MXVR is disabled, the DVALID bit will be 0. When the MXVR is enabled in Master Mode, the DVALID bit will be 1 and the DELAY field will be 0. When the MXVR is enabled in Master Mode and the DVALID bit is 1, only asserting reset or disabling the MXVR will cause the DVALID bit to change to 0. When the MXVR is enabled in Slave Mode, the DVALID bit will be 0 until the MXVR has reached a lock level at which the node delay can be correctly determined from the incoming datastream. Once the node delay is correctly determined, the DVALID bit will change to a 1 and the DELAY field will contain the node delay value. Subsequent changes to the node delay (for example, other nodes changing from 2 frame delays to 0 frame delays) will cause the DELAY field to update, but the DVALID bit will remain a 1 as long as the MXVR remains locked. Once the MXVR is enabled in Slave Mode and the DVALID bit is 1, only asserting reset, disabling the MXVR, or losing lock will cause the DVALID bit to change to 0.

Note that synchronous data received by the MXVR and DMA'ed to L1 or L2 memory is not frame delayed in the process of transferring the data and synchronous data that is DMA'ed from L1 or L2 memory to the MXVR for transmit is not frame delayed in the process of transferring the data.

To determine the actual time delay of data transmitted from L1 or L2 memory of one MXVR node "A" to the L1 or L2 memory of MXVR node "B" can be calculated using one of three formulas:

If (POSITIONA < POSITIONB),

t_{delay} = 2 * (DELAYA - DELAYB) * (1 / Fs)

If (POSITIONA > POSITIONB) and (SDELAYA == "0"),

t_{delay} = 2 * (MDELAY - DELAYA + DELAYB) * (1 / Fs)

If (POSITIONA > POSITIONB) and (SDELAYA == "1"),

t_{delay} = 2 * (MDELAY - DELAYA + DELAYB - 1) * (1 / Fs)

MXVR Maximum Node Frame Delay (MXVR_MAX_DELAY) Register

The MXVR_MAX_DELAY register is a read-only register that indicates the total number of nodes with two frame delays that synchronous data will pass through when circling the network. The total number of node delays is calculated by the Master once every 1024 frames. Then the Master distributes the MDELAY value to all the Slave nodes once every 1024 frames.

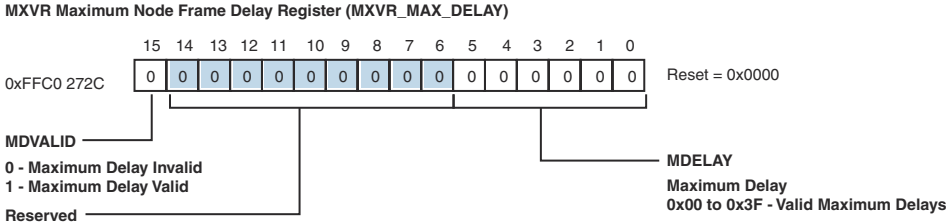


Figure 29-11. MXVR Maximum Node Frame Delay Register

If the MDVALID bit is a 1, then the MDELAY field is valid. If the MDVALID bit is a 0, then the MDELAY field is not valid. The MDELAY field can range from 0 to 63 (representing from 0 to 126 frame delays for synchronous data).

When the MXVR is disabled, the MDVALID bit will be 0. When the MXVR is enabled in Master Mode, the MDVALID bit will be 0 until the Master reaches a lock level at which the total number of node delays in the network can be determined. Once the total number of node delays is correctly determined, the MDVALID bit will change to a 1 and the MDELAY field will contain the total number of node delays. Then the Master will distribute the total number of delays in the network to the Slave nodes once every 1024 frames.

When the MXVR is enabled in Slave Mode, the MDVALID bit will be 0 until the MXVR has reached a lock level at which the total number of node delays can correctly received from the Master. Once the total number of node delays is correctly received, the MDVALID bit will change to a 1 and

the MDELAY field will contain the total number of node delays in the network. Subsequent changes to the total number of node delays (for example, other nodes changing from 2 frame delays to 0 frame delays) will cause the MDELAY field to update, but the MDVALID bit will remain a 1 as long as the MXVR remains locked.

Once MDVALID is set to 1, only asserting reset, disabling the MXVR, or losing lock will cause the MDVALID to change to 0.

MXVR Logical Address (MXVR_LADDR) Register

The MXVR_LADDR register sets the MXVR node's logical address. The logical address may be programmed to any value; however, address 0x0000 is not allowed by the protocol, addresses 0x3000 to 0x03FF are reserved for group and broadcast addresses and addresses 0x0400 to 0x04FF are reserved for position addresses. In addition, software must determine the uniqueness of any logical address.

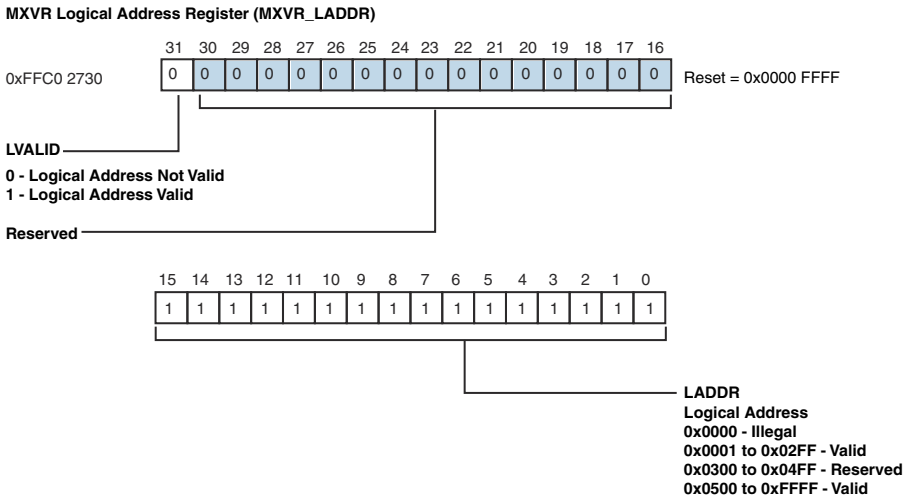


Figure 29-12. MXVR Logical Address Register

There is an LVALID bit which should be written to a 1 once the LADDR field is written. When the LVALID bit is set to a 1, the MXVR will use the value of the LADDR field as the Logical Address for checking the Destination Address field of incoming Asynchronous Packets and Control Messages. If the LVALID bit is set to 0, the MXVR will only use the Alternate Address from the MXVR_AADDR register for checking the Destination Address field of incoming Asynchronous Packets. If the LVALID bit is set to 0, the MXVR will only use the Physical Address from the MXVR_POSITION register and the Group Address from the MXVR_GADDR register for checking the Destination Address field of incoming Control Messages.

MXVR Group Address (MXVR_GADDR) Register

The MXVR_GADDR register sets the MXVR node's group address. This address may be programmed to any value and software must determine the suitability of any group address. The lower byte of the Group Address can be written to the GADDRL field. The upper byte is assumed to be 0x03. There is a GVALID bit which should be written to a 1 once the GADDRL field is written. When the GVALID bit is set to a 1, the MXVR will use the value of the GADDRL field to form the Group Address for checking the Destination Address field of incoming Control Messages. If the GVALID bit is set to 0, the MXVR will only use the Physical Address from the MXVR_POSITION register and the Logical Address from the MXVR_LADDR register for checking the Destination Address field of incoming Control Messages.

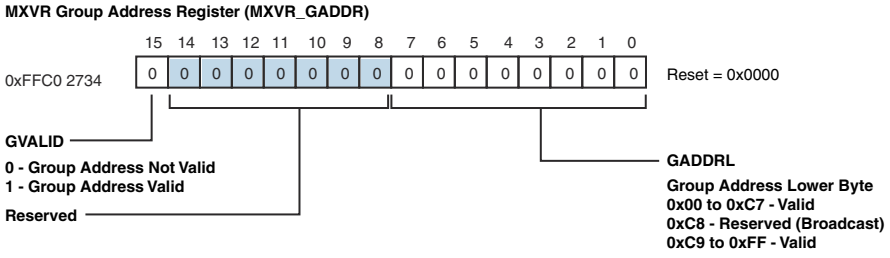


Figure 29-13. MXVR Group Address Register

MXVR Alternate Address (MXVR_AADDR) Register

The MXVR_AADDR register sets the MXVR node's alternate address. The alternate address may be programmed to any value and software must determine the suitability of any alternate address. The AVALID bit should be written to a 1 once the AADDR field is written. When the AVALID bit is set to a 1, the MXVR will use the value of the AADDR field as the Alternate Address for checking the Destination Address field of incoming Asynchronous Packets. If the AVALID bit is set to 0, the MXVR will only use the Logical Address for checking the Destination Address field of incoming Asynchronous Packets.

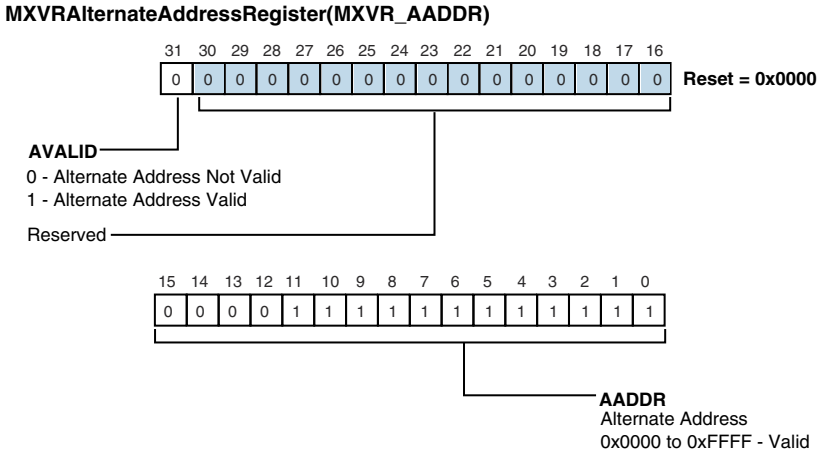


Figure 29-14. MXVR Alternate Address Register

MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers

The MXVR_ALLOC_x registers contain the Allocation Table for the network's synchronous physical channels. The Master services all allocation and de-allocation requests, maintains the complete Allocation Table, and

sends the Allocation Table out to all the Slave nodes once every 1024 frames. All Allocation Table related processing is handled by the MXVR Master in hardware (without interaction from software).

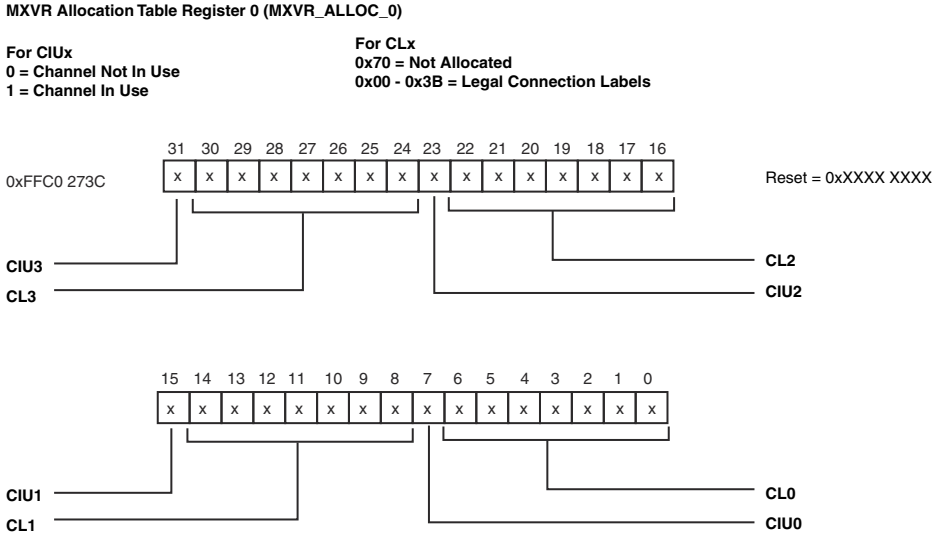


Figure 29-15. MXVR Allocation Table Register

The Allocation Table appears in fifteen read-only registers (MXVR_ALLOC_0 to MXVR_ALLOC_14). The 60 synchronous physical channels each have an 8-bit section in one of the 32-bit MXVR_ALLOC_x registers. Figure 8-17 shows MXVR_ALLOC_0 register as an example of one of the Allocation Table registers. All other Allocation Table registers have the same format.

The Connection Label (CLx) field indicates which physical channels are associated with a particular Connection Label value. When the CLx field value is 0x70, physical channel x has not been allocated. When the CLx field value is between 0x00 and 0x3B, physical channel x is allocated and is associated with all other physical channels which have the same CLx field value.

The Channel-In-Use (CIUx) bit indicates whether a particular physical channel is “In-Use” by a node in the network. If the CIUx bit is 0, physical channel x is not “In-Use”. If the CIUx bit is 1, physical channel x is “In-Use”.

The Master node modifies its Allocation Table based on Resource Allocate and Resource De-Allocate system control messages from itself and from the Slave nodes in the ring. The Master node distributes the Allocation Table to all Slaves in the ring over the control message channel once every 1024 frames. As each Slave node receives the Allocation Table, the Slave node updates its own copy of the Allocation Table and also sets the CIUx bit for each physical channel that Slave node is using. In this way, once the Allocation Table returns back to the Master, the Master’s Allocation Table will show which channels are “In-Use” for the entire network. Note that in each Slave node, the CIUx bits only reflect which channels are “In-Use” by upstream nodes (nodes with lower POSITION values).

MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers

The MXVR_SYNC_LCHAN_x registers are used to assign logical channel numbers to each of the 60 synchronous physical channels. These logical channel numbers are then used when programming the 8 synchronous data DMA channels.

There are eight Synchronous Logical Channel Assignment registers (MXVR_SYNC_LCHAN_0 to MXVR_SYNC_LCHAN_7). The 60 synchronous physical channels each have an 4-bit field in one of the eight 32-bit MXVR_SYNC_LCHAN_x registers. Figure 8-18 shows MXVR_SYNC_LCHAN_0 register as an example of one of the Synchronous Logical Channel Assignment registers. All other Synchronous Logical Channel Assignment registers have the same format.

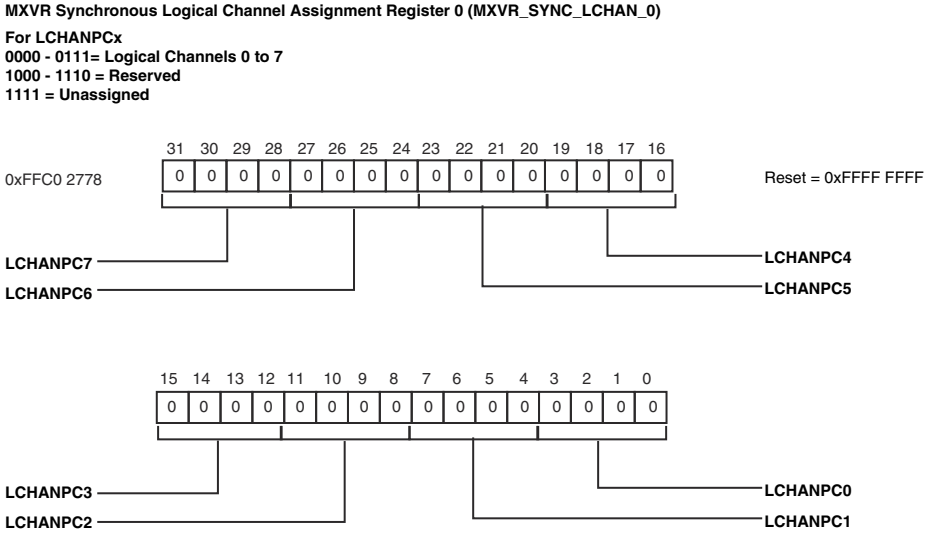


Figure 29-16. MXVR Synchronous Logical Channel Assignment Register

The Logical Channel for Physical Channel x (LCHANPCx) field gives the logical channel number assigned to physical channel x. All LCHANPCx fields will reset to b#1111 which indicates that the physical channel has not been assigned to a logical channel. Each physical channel which will be used for receiving data or transmitting data should have the LCHANPCx field assigned a logical channel value between b#0000 and b#0111. Logical channel values between b#1000 and b#1110 are reserved.

All physical channels which have the same logical channel value programmed to their LCHANPCx fields will be DMA'ed together. For example, if LCHANPC5, LCHANPC8, and LCHANPC30 each have been written with b#0110 and Synchronous DMA Channel 3 is programmed to receive data from logical channel 6 (LCHAN3=b#0110), then Synchronous Data DMA Channel 3 will DMA data received on physical channels 5, 8, and 30 into L1 memory.

Note that the logical channel numbers assigned to the LCHANPCx fields have no meaning other than to associate physical channels with each other and assign them to DMA channels. These logical channel numbers are completely independent of the Connection Label numbers in the Allocation Table.

MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers

The MXVR_DMAx_CONFIG registers set the operating mode for the eight Synchronous Data DMA channels. Each Synchronous Data DMA channel can transfer synchronous data received by the MXVR from the network to L1 or L2 memory or can transfer synchronous data stored in L1 or L2 memory to the MXVR to be transmitted over the network. The physical channels allocated for transferring synchronous data can be grouped into logical channels by programming the MXVR_SYNC_LCHAN_x registers. The Synchronous Data DMA channels can then be assigned to a particular logical channel for transmit or receive. In this way synchronous data can easily be moved from any set of received channels to L1 or L2 memory or from L1 or L2 memory to any set of transmitted channels.

The DMA channel is enabled by setting the DMAx Enable (MDMAENx) bit to 1 or disabled by setting the MDMAEN bit to 0. When the MDMAENx bit is set to 1, the MXVR_DMAx_START_ADDR and MXVR_DMAx_COUNT registers should not be written. In addition when the MDMAENx bit is set to 1, all bits in the MXVR_DMAx_CONFIG register except for the MDMAENx bit will be read-only and writes to other bits in the MXVR_DMAx_CONFIG register will have no effect. Note that when a DMA channel is enabled or disabled, the DMA channel will always start or stop at the beginning of a new frame.

When disabling a DMA channel (by setting MDMAENx to 0), the associated Logical Channel assignment made in the MXVR_SYNC_LCHAN_x registers must not be changed until after the associated DMAACTIVEx bit in the MXVR_STATE_1 register indicates that the DMA channel has stopped.

The transfer direction for the DMA channel is set by writing the DMAx Direction (DDx) bit. When the DDx bit is set to 1, the DMA channel will transfer data received by the MXVR to an L1 or L2 memory buffer. When the DDx bit is set to 0, the DMA channel will transfer data from an L1 or L2 memory buffer to the MXVR to be transmitted.

The DMAx Four Byte Swap Enable (BY4SWAPENx) bit enables or disables four byte swapping of the data that is DMA'ed to/from L1 or L2 memory. If BY4SWAPENx is set to 1, the data byte 0 will be swapped with data byte 3 and data byte 1 will be swapped with data byte 2. If BY4SWAPENx is set to 0, four byte swapping will not take place. For example, data value 0x54987536 when four byte swapped becomes 0x36759854. Four byte swapping is done by reading and writing the L1 or L2 memory in a different order if four byte swapping is enabled. For example, normally data will be read from/written to L1 or L2 in the following address order: 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, etc. If four byte swapping is enabled, data will be read from/written to L1 or L2 in the following address order: 0x03, 0x02, 0x01, 0x00, 0x07, 0x06, 0x05, 0x04, etc. Note that when four byte swapping is enabled, the MXVR_DMAx_CURR_ADDR will reflect the normal address incrementing (0x00, 0x01, 0x02, 0x03, etc.) even though the L1 or L2 memory accesses will be occurring in the four byte swapping address order. Note that bit-swapping and four byte-swapping may be used in conjunction. However, two byte-swapping and four byte-swapping may not be used at the same time.

The DMAx Logical Channel (LCHANx) field determines which logical channel in the incoming frame will be received and DMA'ed to L1 or L2 memory or which logical channel in the outgoing frame will be DMA'ed from L1 or L2 memory and transmitted. The logical channels are defined in the MXVR_SYNC_LCHAN_x registers. Two DMA channels can have the same LCHANx field set as long as the data direction for the two channels is different (one for receive, one for transmit). Programming more than one DMA channel with the same data direction and the same LCHANx value is illegal.

The DMAx Bit-Swap Enable (BITSWAPENx) bit enables or disables bit swapping of the data that is DMA'ed to and from L1 memory. If BITSWAPENx is set to 1, the data bits will be swapped on a byte-wise basis as follows:

bit 7 => bit 0 and bit 0 => bit 7

bit 6 => bit 1 and bit 1 => bit 6

bit 5 => bit 2 and bit 2 => bit 5

bit 4 => bit 3 and bit 3 => bit 4

For example, data value 0x35 when bit-swapped becomes 0xAC. If BITSWAPEN is set to 0, no bit swapping will take place. Note that bit-swapping and byte-swapping may be used in conjunction.

The DMAx Two Byte Swap Enable (BY2SWAPENx) bit enables or disables two byte swapping of the data that is DMA'd to/from L1 or L2 memory. If BY2SWAPENx is set to 1, the data byte 0 will be swapped with data byte 1. If BY2SWAPENx is set to 0, two byte swapping will not take place. For example, data value 0x3586 when two byte swapped becomes 0x8635. Two byte swapping is done by reading and writing the L1 or L2 memory in a different order if two byte swapping is enabled. For example, normally data will be read from/written to L1 or L2 in the following address order: 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, etc. If two byte swapping is enabled, data will be read from/written to L1 or L2 in the following address order: 0x01, 0x00, 0x03, 0x02, 0x05, 0x04, etc. Note that when two byte swapping is enabled, the MXVR_DMAx_CURR_ADDR will reflect the normal address incrementing (0x00, 0x01, 0x02, 0x03, etc.) even though the L1 or L2 memory accesses will be occurring in the two byte swapping address order. Note that bit-swapping and two byte-swapping may be used in conjunction. However, two byte-swapping and four byte-swapping may not be used at the same time.

The DMAx Operation Flow (MFLOWx) field determines the operating mode of the DMA channel. Each DMA channel can operate in Stop Mode, Autobuffer Mode, Synchronous Packet-Fixed Count Mode, Synchronous Packet-Variable Count Mode, and Synchronous Packet-Start/Stop Mode.

In Stop Mode, once the DMA is enabled a fixed number of bytes of data will be transferred from the logical channel to an L1 or L2 memory buffer (receive) or from an L1 or L2 memory buffer to the logical channel (transmit). The starting address of the L1 or L2 memory buffer is programmed in the MXVR_DMAx_START_ADDR register and the number of bytes to be transferred is programmed in the MXVR_DMAx_COUNT register. The DMA channel will set the HDONEx status event when half of the total number of bytes are completed, and will set the DONEx status event when the total number of bytes are completed. Once all the transfers are done the DMA channel will disable itself. Disabling the DMA channel manually before the DMA has completed the total number of bytes will halt the DMA transfers and the values in the MXVR_DMAx_CURR_ADDR and MXVR_DMAx_CURR_COUNT will indicate where the DMA channel stopped. However, when the channel is re-enabled, the current address and count will reset back to the values programmed into the MXVR_DMAx_START_ADDR and MXVR_DMAx_COUNT.

In Autobuffer Mode, once the DMA is enabled a fixed number of bytes of data will be transferred from the logical channel and to an L1 or L2 memory buffer (receive) or from an L1 or L2 memory buffer to the logical channel (transmit). The starting address of the L1 or L2 memory buffer is programmed in the MXVR_DMAx_START_ADDR register and the number of bytes to be transferred is programmed in the MXVR_DMAx_COUNT register. The DMA channel will set the HDONEx status event when half of the total number of bytes are completed, and will set the DONEx status event when the total number of bytes are completed. Once all the transfers are done the DMA will remain enabled and will restart from the address specified in the MXVR_DMAx_START_ADDR register and with the transfer count in the MXVR_DMAx_COUNT register. Disabling the DMA channel manually when the DMA is programmed for Autobuffer Mode will halt the DMA transfers and the values in the MXVR_DMAx_CURR_ADDR and

MXVR_DMAx_CURR_COUNT will indicate where the DMA channel stopped. However, when the channel is re-enabled, the current address and count will reset back to the values programmed into the MXVR_DMAx_START_ADDR and MXVR_DMAx_COUNT.

The DMA channels have three Synchronous Packet Autobuffer Modes which allow the DMA channels to receive packetized data over the synchronous data channels. The three modes are Synchronous Packet-Variable Count Mode, Synchronous Packet-Start/Stop Mode, and Synchronous Packet-Fixed Count Mode. These DMA modes are only used when the MXVR is receiving data and the DMA channel is writing the data to L1 or L2 memory. These Synchronous Packet Autobuffer Modes allow the data being received to trigger the DMA channel to start at the beginning of a packet and trigger the DMA channel to stop at the end of the packet. Note that the Synchronous Packet Autobuffer Modes which allow the DMA channels to receive packets of data over the synchronous data portion of the network frame should not be confused with Asynchronous Packets which are transmitted and received over the asynchronous data portion of the network frame.

When the DMA channel is set for Synchronous Packet-Variable Count Mode and once the DMA channel is enabled, the DMA channel will search the data in a logical channel in the received data stream for the “start pattern”. The logical channel which the DMA channel will search in and DMA from is defined by the LCHANx field and the “start pattern” is selected by the STARTPATx field. Once the “start pattern” is found, the DMA channel will start transferring data received in the logical channel to L1 or L2 memory and at the same time will search for the transfer count (a 16-bit value representing the number of bytes to be transferred) in the logical channel data stream. The position of the transfer count with respect to the “start pattern” is programmed in the COUNTPOSx field. The MXVR_DMAx_CURR_COUNT will initially be set to 0xFFFF when the “start pattern” is found and will decrement with every transfer done prior to receiving the transfer count. Once the transfer count is received, the MXVR_DMAx_CURR_COUNT will be based on transfer count from the

datastream. Once the DMA channel transfers the number of bytes based on the transfer count to L1 or L2 memory, the DMA will stop transferring data. The DMA channel will then repeat the process and start looking for the “start pattern” again.

The first packet of data (and subsequent odd packet numbers) received will be written to the address specified in the MXVR_DMAx_START_ADDR. The DMA transfers will continue until the transfer count expires. When the transfer count expires, the HDONEx bit in the MXVR_INT_STAT_1 register will be set to 1. The second packet of data (and subsequent even packet numbers) received will be written to an address that is defined by the MXVR_DMAx_START_ADDR plus the value programmed in the MXVR_DMAx_COUNT. The DMA transfers will continue until the transfer count expires. When the transfer count expires, the DONEx bit in the MXVR_INT_STAT_1 register will be set to 1. Subsequent received packets will ping-pong between these two L1 or L2 memory buffers. Note that the value programmed to the MXVR_DMAx_COUNT should be sufficiently large enough to accommodate the largest packet size that will be received.

Synchronous Packet-Variable Count Mode operation will continue until the MDMAENx is set to 0, until a DMA Out of Bounds Error occurs, or until a DMA Error occurs. Note that when a DMA channel is enabled or disabled, the DMA channel will always start or stop at the beginning of a new frame.

When the DMA channel is set for Synchronous Packet-Start/Stop Mode and once the DMA channel is enabled, the DMA channel will be searching the data in a logical channel in the received data stream for the “start pattern”. The logical channel which the DMA channel will search in and DMA from is defined by the LCHANx field and the “start pattern” is selected by the STARTPATx field. Once the “start pattern” is found, the DMA channel start transferring data received in the logical channel to L1 or L2 memory and at the same time will search for the “stop pattern” in the logical channel data stream. The “stop pattern” is selected by the STOPPATx field. The MXVR_DMAx_CURR_COUNT will initially be set to 0xFFFF

when the “start pattern” is found and will decrement with every transfer done prior to receiving “stop pattern”. Once the DMA channel receives the “stop pattern”, the DMA will stop transferring data. The DMA channel will then repeat the process and start looking for the “start pattern” again.

The first packet (and subsequent odd packet numbers) of data received will be written to the address specified in the MXVR_DMAx_START_ADDR. The DMA transfers will continue until the “stop pattern” is found. Once the “stop pattern” is found, the HDONEx bit in the MXVR_INT_STAT_1 register will be set to 1. The second packet (and subsequent even packet numbers) of data received will be written to an address that is defined by the MXVR_DMAx_START_ADDR plus the value programmed in the MXVR_DMAx_COUNT. The DMA transfers will continue until the “stop pattern” is found. Once the “stop pattern” is found, the DONEx bit in the MXVR_INT_STAT_1 register will be set to 1. Received packets will ping-pong between these two L1 or L2 memory buffers. Note that the value programmed to the MXVR_DMAx_COUNT should be sufficiently large enough to accommodate the largest packet size that will be received.

The Synchronous Packet-Start/Stop Mode operation will continue until the MDMAENx is set to 0, until a DMA Out of Bounds Error occurs, or until a DMA Error occurs. Note that when a DMA channel is enabled or disabled, the DMA channel will always start or stop at the beginning of a new frame.

When the DMA channel is set for Packet-Fixed Count Mode and once the DMA channel is enabled, the DMA channel will be searching the data in a logical channel in the received data stream for the “start pattern”. The logical channel which the DMA channel will search in and DMA from is defined by the LCHANx field and the “start pattern” is selected by the STARTPATx field. Once the “start pattern” is found, the DMA channel starts transferring data received in the logical channel to L1 or L2 memory using the transfer count programmed in the MXVR_DMAx_COUNT register (the fixed transfer count). Once the DMA channel transfers the number of

bytes based on the transfer count to L1 or L2 memory, the DMA will stop transferring data. The DMA channel will then repeat the process and start looking for the “start pattern” again.

The first packet (and subsequent odd packet numbers) of data received will be written to the address specified in the MXVR_DMAx_START_ADDR. The DMA transfers will continue until the transfer count expires. Once the transfer count expires, the HDONEx bit in the MXVR_INT_STAT_1 register will be set to 1. The second packet (and subsequent even packet numbers) received will be written to an address that is defined by the MXVR_DMAx_START_ADDR plus the value programmed in the MXVR_DMAx_COUNT. The DMA transfers will continue until the transfer count expires. When the transfer count expires, the DONEx bit in the MXVR_INT_STAT_1 register will be set to 1. Received packets will ping-pong between these two L1 or L2 memory buffers.

The Synchronous Packet-Fixed Count Mode operation will continue until the MDMAENx is set to 0 or until a DMA Error occurs. Note that when a DMA channel is enabled or disabled, the DMA channel will always start or stop at the beginning of a new frame.

The Fixed Pattern Matching select (FIXEDPM) bit determines whether a pattern match can occur on any byte or bytes in a logical channel or if the pattern must match the first byte or bytes of the logical channel. If the FIXEDPM is set to 0, the “start pattern” or “stop pattern” can match any byte or bytes in the logical channel. If the FIXEDPM is set to 1, the “start pattern” or “stop pattern” will only match if the first byte of the pattern matches the first byte in the logical channel (and so on depending on how many bytes are being matched). For example, if the pattern is two bytes long and the logical channel is defined as physical channels 8 to 11 and if FIXEDPM is set to 1, then byte 0 of the pattern must match physical channel 8 and byte 1 of the pattern must match physical channel 9 for there to be a match. In the same example if FIXEDPM is set to 0, bytes 0 and 1 could

match physical channels 8 and 9, 9 and 10, 10 and 11, 11 in the current frame and 8 in the next frame, or 11 from the previous frame and 8 in the current frame.

The Start Pattern select (STARTPATx) field determines which set of pattern registers will specify the “start pattern”. If the STARTPATx is set to b#00, pattern registers MXVR_PAT_DATA_0 and MXVR_PAT_EN_0 will specify the “start pattern”. If the STARTPAT is set to b#01, pattern registers MXVR_PAT_DATA_1 and MXVR_PAT_EN_1 will specify the “start pattern”. All other values of STARTPATx are reserved. Note that the “start pattern” itself will not be DMA'd to L1 or L2 memory.

The Stop Pattern select (STOPPATx) field determines which set of pattern registers will specify the “stop pattern”. If the STOPPATx is set to b#00, pattern registers MXVR_PAT_DATA_0 and MXVR_PAT_EN_0 will specify the “stop pattern”. If the STOPPATx is set to b#01, pattern registers MXVR_PAT_DATA_1 and MXVR_PAT_EN_1 will specify the “stop pattern”. All other values of STOPPATx are reserved. Note that the “stop pattern” itself will be DMA'd to L1 or L2 memory.

The Count Position (COUNTPOSx) field indicates where the 16-bit transfer count can be found in the received data stream once the “start pattern” is found when operating in Synchronous Packet-Variable Count Mode. The COUNTPOSx indicates the position of the transfer count by giving the number of bytes between the last byte of the “start pattern” to the first byte of the transfer count. The COUNTPOSx can range from 0 bytes after the end of the “start pattern” to 7 bytes after the end of the “start pattern.” For example, if the COUNTPOSx was set to 0, then the transfer count would be found in the first two bytes in the logical channel after the end of the “start pattern”. If the COUNTPOSx was set to 7, then the transfer count would be found in the eight and ninth bytes in the logical channel after the end of the “start pattern”. The most significant byte of the transfer count is received first, and followed by the least significant byte of the transfer count.

Note that the number of bytes set by the COUNTPOSx field is with respect to the logical channel data stream. In other words, the “start pattern” and transfer count will be in the same logical channel but may be in different frames. Note that the bytes of data between the “start pattern” and the transfer count, and the transfer count itself will be DMA'd to L1 or L2 memory.

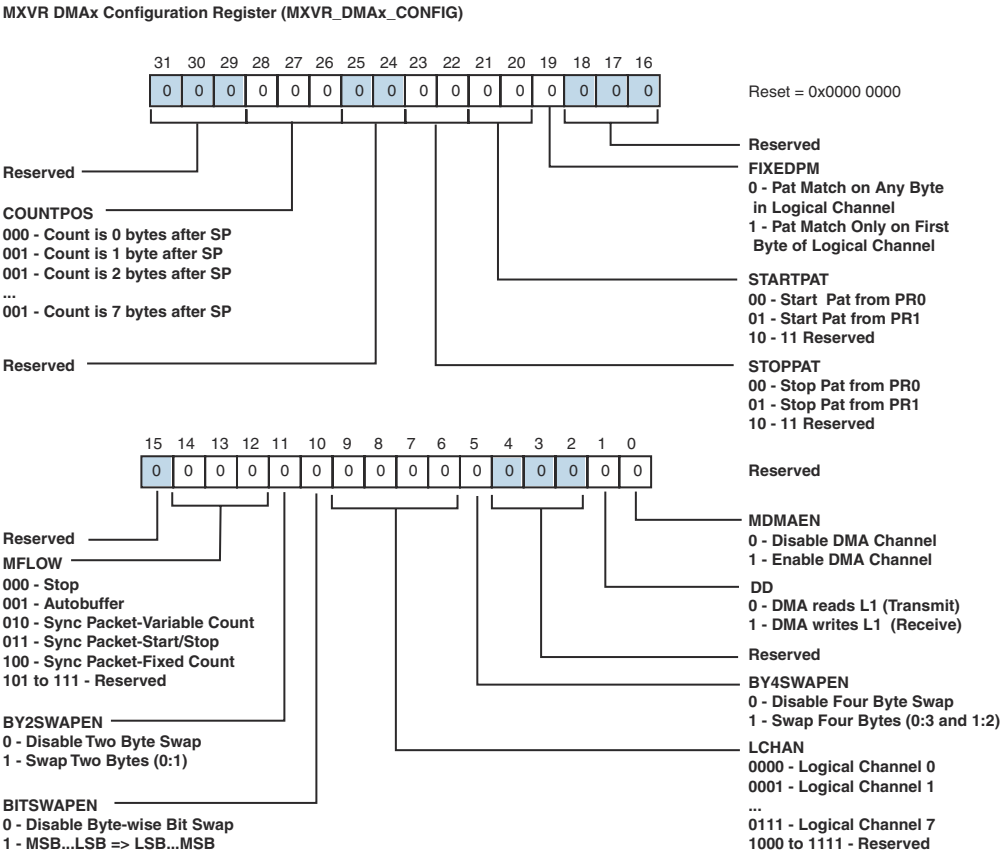


Figure 29-17. MXVR DMAx Configuration Register

MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers

The MXVR_DMAx_START_ADDR registers set the starting address for the synchronous data DMA channels. The synchronous data DMA channels can only DMA to or from L1 or L2 memory. Therefore, bits 31-25 are fixed to 1s.

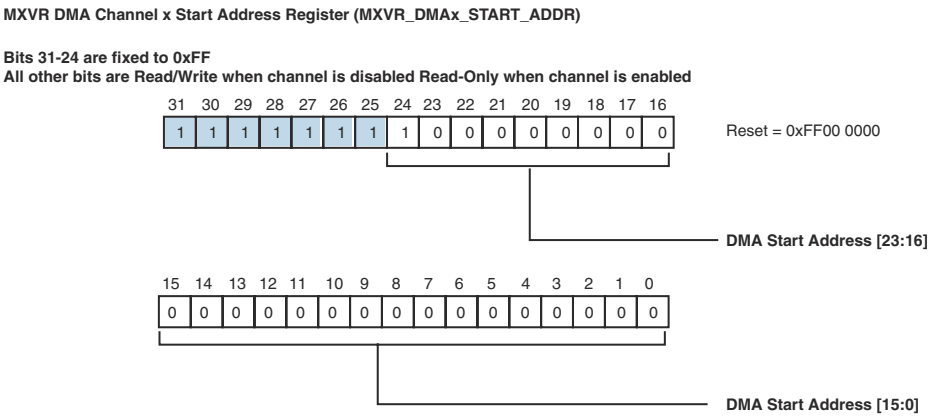


Figure 29-18. MXVR DMA Channel x Start Address Registers

Once the DMA is enabled, data will begin to be DMA'd to or from the address given in the MXVR_DMAx_START_ADDR for that channel. The operation of the DMA channel depends on which DMA mode is selected with the MFLOWx field:

If the DMA is operating in Stop Mode, once all the transfers specified in the corresponding MXVR_DMAx_COUNT have been done, the DMA will automatically disable.

If the DMA channel is operating in Autobuffer Mode, once all of the transfers specified in the corresponding MXVR_DMAx_COUNT have been done, the DMA will then jump back to the start address programmed in the

MXVR_DMAx_START_ADDR and DMA operations will continue from there. In this way the data received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT/2.

If the DMA channel is operating in Synchronous Packet-Variable Count Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the transfer count field in the packet itself have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the transfer count field in the packet itself have been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

If the DMA channel is operating in Synchronous Packet-Start/Stop Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the amount of data received between the "start pattern" and the "stop pattern" have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the amount of data received between the "start pattern" and the "stop pattern" have been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

If the DMA channel is operating in Synchronous Packet-Fixed Count Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the fixed count in the MXVR_DMAx_COUNT have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the fixed count in the MXVR_DMAx_COUNT have

been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers

The MXVR_DMAx_CURR_ADDR registers are read-only registers which give the current address that the synchronous data DMA channels are accessing. The synchronous data DMA channels can only DMA to or from L1 or L2 memory. Therefore, bits 31–25 are fixed to 1s. Once the DMA is enabled, data will begin to be DMA'd to or from the address given in the MXVR_DMAx_START_ADDR for that channel. The MXVR_DMAx_CURR_ADDR will always show the address which is being DMA'd to or from or the address that was DMA'd to or from previously for each channel.

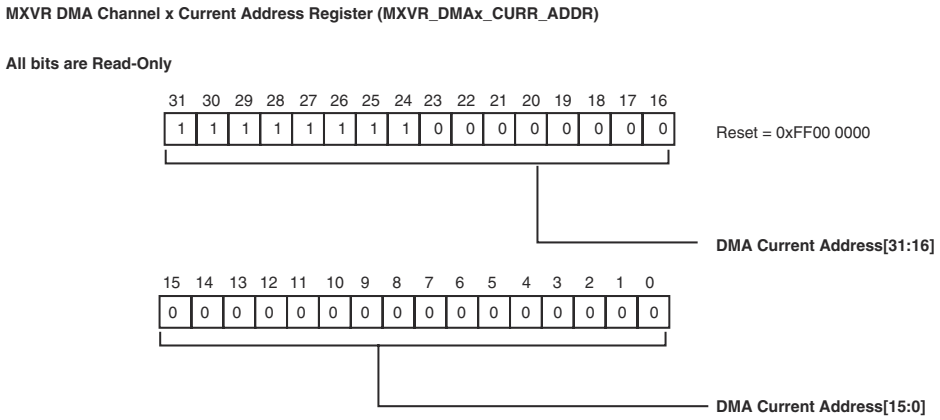


Figure 29-19. MXVR DMA Channel x Current Address Register

MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers

The MXVR_DMAx_COUNT registers set the number of bytes that the synchronous data DMA channels will transfer. The synchronous data DMA channels can only DMA to or from L1 or L2 memory. The maximum MXVR_DMAx_COUNT value is 65535 (giving a maximum data block size to be DMA'd of 64K bytes). The value 0x0000 is illegal and should not be written to the MXVR_DMAx_COUNT register.

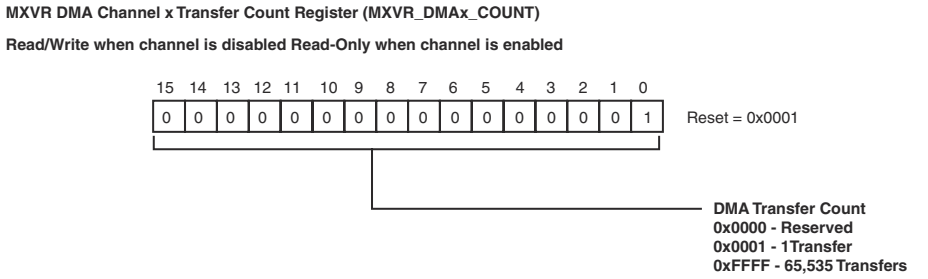


Figure 29-20. MXVR DMA Channel x Transfer Count Registers

Once the DMA is enabled, data will begin to be DMA'd to or from the address given in the MXVR_DMAx_START_ADDR for that channel. The meaning of the MXVR_DMAx_COUNT and the operation of the DMA channel depend on which DMA mode is selected with the MFLOWx field:

If the DMA is operating in Stop Mode, the MXVR_DMAx_COUNT value is the total number of bytes to be transferred. Once half of the transfers specified have been completed, the HDONE interrupt event will be generated. Once all the transfers specified have completed, the DONE interrupt event will be generated and the DMA will automatically be disabled.

If the DMA channel is operating in Autobuffer Mode, the MXVR_DMAx_COUNT value is the total number of bytes to be transferred before the address is reset back to the MXVR_DMAx_START_ADDR. Once half of the transfers specified have completed, the HDONE interrupt event will be generated. Once all the transfers specified have completed, the DONE interrupt event will be generated and the DMA will jump back to the start address programmed in the MXVR_DMAx_START_ADDR and DMA operations will continue from there.

If the DMA channel is operating in Synchronous Packet-Variable Count Mode, the MXVR_DMAx_COUNT value is the offset from the MXVR_DMAx_START_ADDR where every other packet will be written to. The first packet (third packet, fifth packet, seventh packet, etc.) received will be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The transfer count for each packet is included in the packet itself. Therefore, the MXVR_DMAx_COUNT value should be larger than the length of the largest packet to be received.

If the DMA channel is operating in Synchronous Packet-Start/Stop Mode, the MXVR_DMAx_COUNT value is the offset from the MXVR_DMAx_START_ADDR where every other packet will be written to. The first packet (third packet, fifth packet, seventh packet, etc.) received will be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The number of transfers to be done for each packet is determined by the packet itself based on the number of bytes between the “start pattern” and the “stop pattern”. Therefore, the MXVR_DMAx_COUNT value should be sufficiently large to hold the longest packet to be received.

If the DMA channel is operating in Synchronous Packet-Fixed Count Mode, the MXVR_DMAx_COUNT value is the number of bytes that will be transferred to store one packet. All packets will be of the same length. The first packet (third packet, fifth packet, seventh packet, etc.) received will

be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers

The MXVR_DMAx_CURR_COUNT registers are read-only registers which give an indication of the current number of bytes remaining to be transferred for that synchronous data DMA channel. The meaning of the value in the MXVR_DMAx_CURR_COUNT depends which DMA mode is selected with the MFLOWx field.

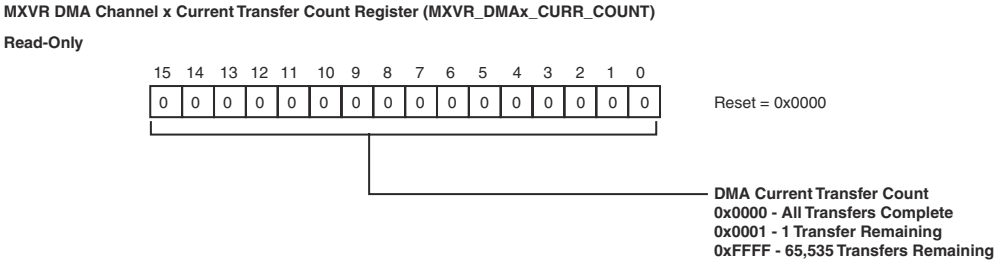


Figure 29-21. MXVR DMA Channel x Current Transfer Count Registers

In Stop Mode, Autobuffer Mode, and Synchronous Packet-Fixed Count Mode, the MXVR_DMAx_CURR_COUNT will always show the number of bytes which still need to be transferred. When all the transfers that were specified are done, the MXVR_DMAx_CURR_COUNT will be 0x0000.

In Synchronous Packet-Variable Count Mode, the number of bytes to be transferred is not known until the transfer count is found in the packet. Therefore, prior to finding the transfer count the MXVR_DMAx_CURR_COUNT will decrement from 0xFFFF. Once the transfer count is found, the

MXVR_DMAx_CURR_COUNT will show the number of bytes which still need to be transferred. When all the transfers that were specified are done, the MXVR_DMAx_CURR_COUNT will be 0x0000.

In Synchronous Packet-Start/Stop Mode, the number of bytes to be transferred is not known until the “stop pattern” is found. Therefore, the MXVR_DMAx_CURR_COUNT will decrement from 0xFFFF.and will stop when the “stop pattern” is found.

MXVR Asynchronous Packet Control (MXVR_AP_CTL) Register

The MXVR_AP_CTL register is a 16-bit register that is used to control the transmission and reception of Asynchronous Packets. The MXVR has an Asynchronous Packet Transmit Buffer (APTB) and an Asynchronous Packet Receive Buffer (APRB). The APRB is capable of holding two received Asynchronous Packets.

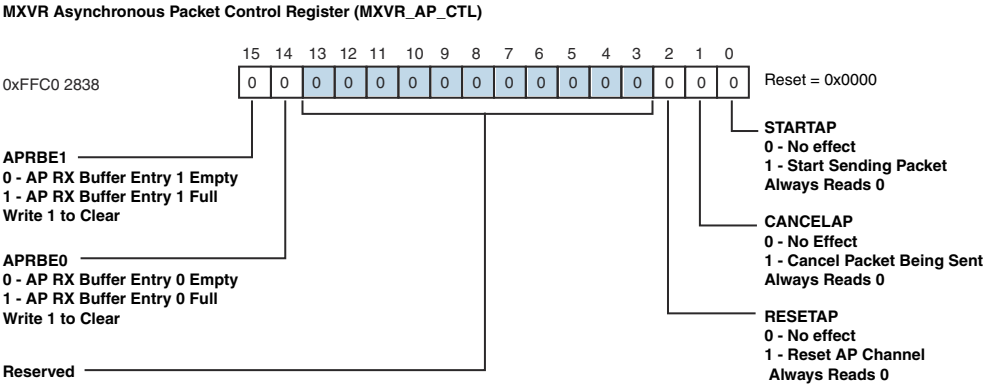


Figure 29-22. MXVR Asynchronous Packet Control Register

The Start Asynchronous Packet Transmission (STARTAP) bit should be set to 1 once an asynchronous packet to be transmitted is written to the APTB and the asynchronous packet is ready to be sent. Once the STARTAP bit is

set to 1, arbitration for the asynchronous channel begins and continues until arbitration is won or until the asynchronous packet is cancelled with the CANCELAP bit. The STARTAP bit always reads as 0 and writing a 0 to the STARTAP bit has no effect.

The Cancel Asynchronous Packet Transmission (CANCELAP) bit allows an asynchronous packet transmission which is arbitrating for the asynchronous channel to be cancelled. Once arbitration is won, the asynchronous packet being sent cannot be cancelled. To cancel the asynchronous packet transmission, the CANCELAP bit should be set to 1. Writing a 1 to the CANCELAP bit after arbitration is won and the asynchronous packet is already being sent will have no effect. The CANCELAP bit always reads as 0 and writing a 0 to the CANCELAP bit has no effect.

The Reset Asynchronous Packet Arbitration (RESETAP) bit allows the Master to reset the asynchronous packet arbitration if an Asynchronous Packet Error (APRPE) is detected. Asynchronous packet errors can occur when the arbitration mechanism gets hung due to a bit error or when the transmitting node does not properly terminate its asynchronous packet transmission (for example, if a node is reset or disabled during an asynchronous packet transmission).

Before the Master asserts the RESETAP, the Master should allow enough time for all nodes in the ring to recognize that an asynchronous packet error has occurred or the Master should notify all slave nodes in the ring that it will be resetting the asynchronous packet arbitration, so that no node will attempt transmission during the reset. The asynchronous packet arbitration reset can take up to 3 frames to complete. The Master should notify the slave nodes in the ring that the reset of the asynchronous packet arbitration has completed.

Resetting the asynchronous packet arbitration while a packet is being transmitted will block the packet from being received by nodes with positions less than the position of the transmitting node. Transmitting

asynchronous packets while the Master is resetting the asynchronous packet arbitration could cause packet collisions and could cause further packet errors.

To reset the asynchronous packet arbitration, the RESETAP bit should be set to 1. Only the Master (MMSM = 1) can cause a reset of the asynchronous packet arbitration. Attempting to write the RESETAP bit to a 0 in a Master node will have no effect. In a Slave node, the RESETAP bit will always be set to 0. Attempting to write the RESETAP bit to a 1 or 0 in a Slave node will have no effect.

The Asynchronous Packet Receive Buffer Entry x (APRBEx) bits indicate whether entry x in the APRB is full or empty. The APRBEx bits are sticky bits which must be written with a 1 to clear. Writing a 0 to the APRBEx bit will have no effect. When a received asynchronous packet is DMA'd to an APRB entry, the corresponding APRBEx bit will be set to 1. Once software has read the Asynchronous Packet stored in that entry, a 1 should be written to the corresponding APRBEx bit in order to clear the bit and to indicate that the entry is empty and can be used for another incoming asynchronous packet. The MXVR will always attempt to DMA an incoming asynchronous packets to the next sequential APRB buffer entry (first asynchronous packet to APRBE0, second to APRBE1, third to APRBE0, etc.). An overflow will occur if the next sequential APRBEx bit is 1 when a new asynchronous packet is being received, and the APROF bit in the MXVR_INT_STAT_0 register will be set to 1.

MXVR Asynchronous Packet Receive Buffer Start Address (MXVR_APRB_START_ADDR) Register

The MXVR_APRB_START_ADDR register sets the starting address for the Asynchronous Packet Receive Buffer in L1 or L2 memory. The APRB must be allocated 2048 bytes. The APRB can only reside in L1 or L2 memory and the APRB must be word aligned. Therefore, bits 31-25 are fixed to 1s and bit 0 is fixed to 0.

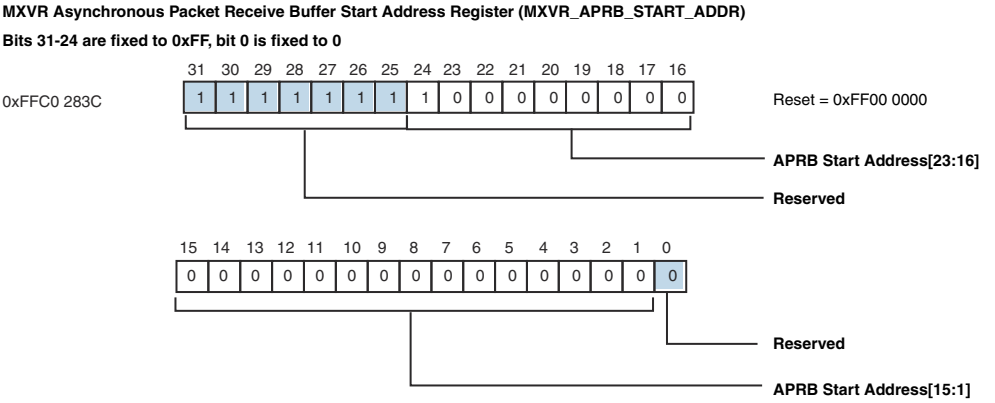


Figure 29-23. MXVR APRB Start Address Register

MXVR Asynchronous Packet Receive Buffer Current Address (MXVR_APRB_CURR_ADDR) Register

The MXVR_APRB_CURR_ADDR register is a read-only register which gives the current address that the Asynchronous Packet Receive DMA channel is writing to in the APRB. The APRB can only reside in L1 or L2 memory. Therefore, bits 31-25 will always be 1's and bit 0 will always be 0.

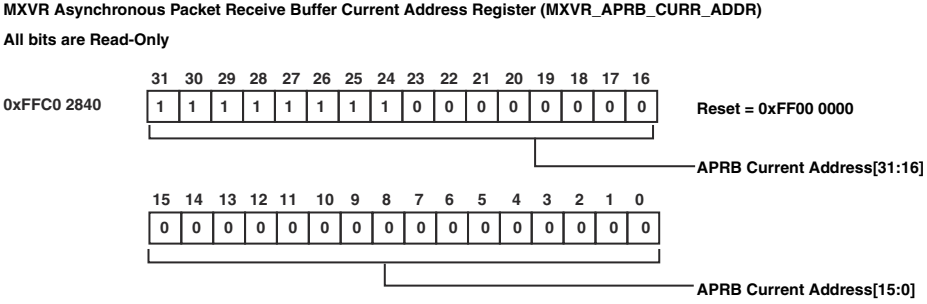


Figure 29-24. MXVR Asynchronous Packet Receive Buffer Current Address Register

MXVR Asynchronous Packet Transmit Buffer Start Address (MXVR_APTB_START_ADDR) Register

The MXVR_APTB_START_ADDR register sets the starting address for the Asynchronous Packet Transmit Buffer in L1 or L2 memory. Enough memory should be allocated to the APTB based on the largest packet to be transmitted. The APTB can only reside in L1 or L2 memory and the APTB must be word aligned. Therefore, bits 31-25 are fixed to 1s and bit 0 is fixed to 0.

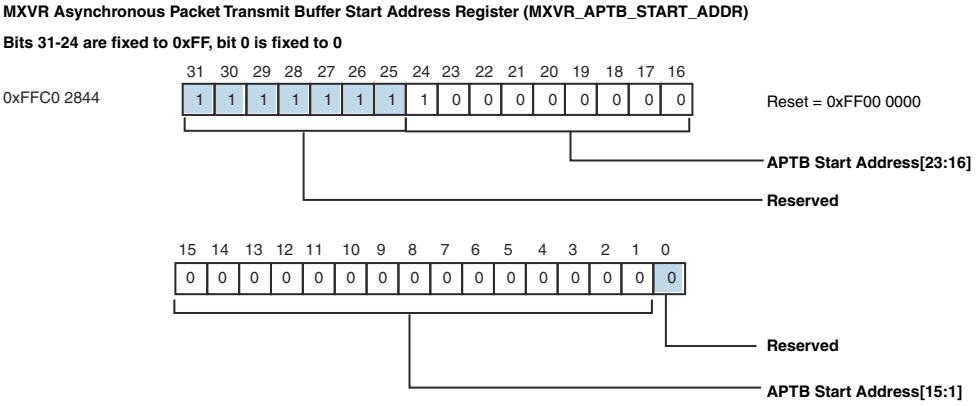


Figure 29-25. MXVR Asynchronous Packet Transmit Buffer Start Address Register

MXVR Asynchronous Packet Transmit Buffer Current Address (MXVR_APTB_CURR_ADDR) Register

The MXVR_APTB_CURR_ADDR register is a read-only register which gives the current address that the Asynchronous Packet Transmit DMA channel is reading from in the APTB. The APTB can only reside in L1 or L2 memory. Therefore, bits 31-24 will always be 1s, and bit 0 will always be 0.

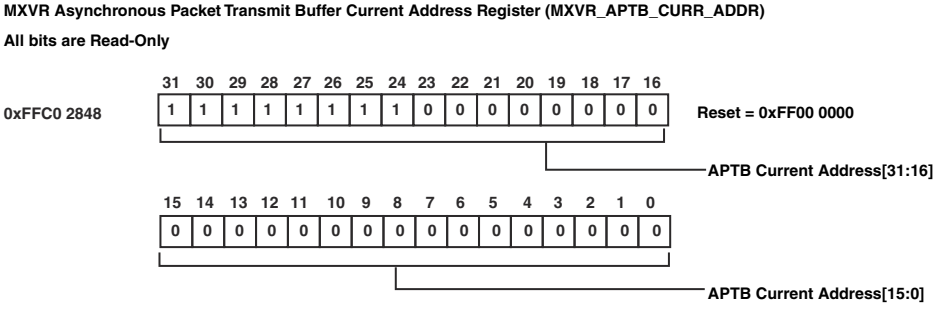


Figure 29-26. MXVR Asynchronous Packet Transmit Buffer Current Address Register

MXVR Control Message Control (MXVR_CM_CTL) Register

The MXVR_CM_CTL register is a 32-bit register that is used to control the transmission and reception of control messages. The MXVR uses a Control Message Transmit Buffer (CMTB) which resides in L1 or L2 memory and holds one control message (System or Normal) to be transmitted. The MXVR also uses a Control Message Receive Buffer (CMRB) which resides in L1 or L2 memory and holds up to 16 received Normal control messages.

The Start Control Message Transmission (STARTCM) bit should be set to 1 when a control message is written to the CMTB and the control message is ready to be sent. Once the STARTCM bit is set to 1, arbitration for the control message channel begins and continues until arbitration is won for the control message to be sent or until the control message is cancelled with the CANCELCM bit. The STARTCM bit always reads as 0 and writing a 0 to the STARTCM bit has no effect.

The Cancel Control Message Transmission (CANCELCM) bit allows a control message (System or Normal) which is arbitrating for the control message channel to be cancelled. Once arbitration is won, the control message being sent cannot be cancelled. To cancel the control message transmission, the CANCELCM bit should be set to 1. Writing a 1 to the CANCELCM bit after arbitration is won and the control message is already being sent will have no effect. The CANCELCM bit always reads as 0 and writing a 0 to the CANCELCM bit has no effect.

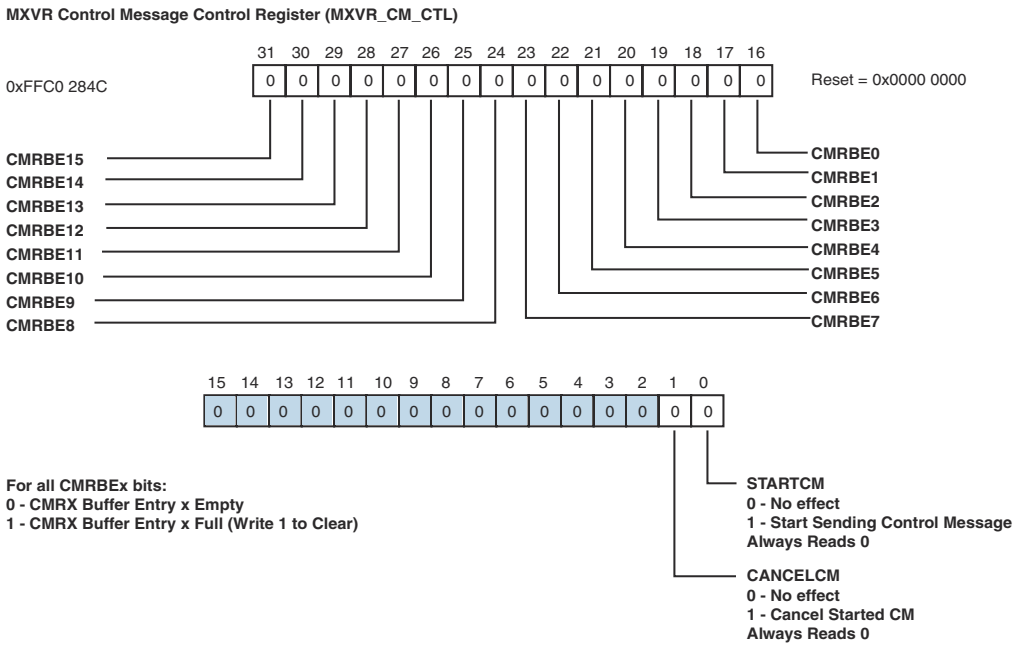


Figure 29-27. MXVR Control Message Control Register

The Control Message Receive Buffer Entry x (CMRBEx) bits indicate whether entry x in the CMRB is full or empty. The CMRBEx bits are sticky bits which must be written with a 1 to clear. When a received Normal control message is DMA'd to the CMRB entry, the corresponding CMRBEx bit will be set to 1. Once software has read the Normal control message stored in that

entry, a 1 should be written to the corresponding CMRBEx bit in order to clear the bit and to indicate that the entry is empty and can be used for another incoming Normal control message. The MXVR will always attempt to DMA an incoming Normal control message to the next sequential CMRB entry (first Normal control message to CMRBE0, second to CMRBE1, …, sixteenth to CMRBE15, seventeenth to CMRBE0, etc.). An overflow will occur if the next sequential CMRBEx bit is 1 when a new Normal control message is arriving, and the CMRBOF bit in the MXVR_INT_STAT_0 register will be set to 1. In addition, when an overflow occurs the Transmission Status will be returned to the transmitter indicating "Receive Buffer Full".

MXVR Control Message Receive Buffer Start Address (MXVR_CMRB_START_ADDR) Register

The MXVR_CMRB_START_ADDR register sets the starting address for the Control Message Receive Buffer (CMRB) in L1 or L2 memory. The CMRB must be allocated 384 bytes. The CMRB can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 are fixed to 1s and bit 0 is fixed to 0.

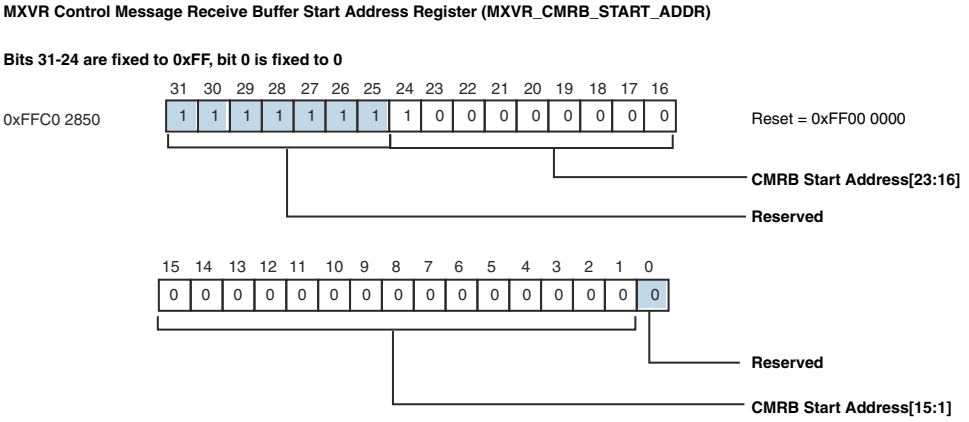


Figure 29-28. MXVR Control Message Receive Buffer Start Address Register

MXVR Control Message Receive Buffer Current Address (MXVR_CMRB_CURR_ADDR) Register

The `MXVR_CMRB_CURR_ADDR` register is a read-only register which gives the current address that the Normal Control Message Receive DMA channel is writing to in the `CMRB`. The `CMRB` can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 will always be 1s and bit 0 will always be 0.

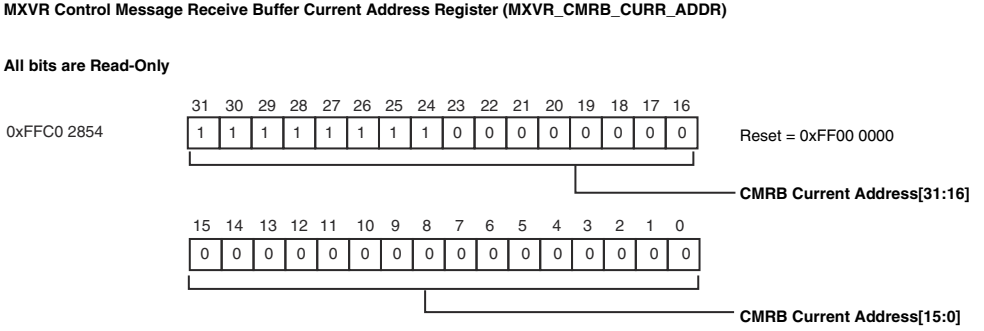


Figure 29-29. MXVR Control Message Receive Buffer Current Address Register

MXVR Control Message Transmit Buffer Start Address (MXVR_CMTB_START_ADDR) Register

The MXVR_CMTB_START_ADDR register sets the starting address for the Control Message Transmit Buffer (CMTB) in L1 or L2 memory. The CMTB must be allocated 26 bytes. The CMTB can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 are fixed to 1s and bit 0 is fixed to 0.

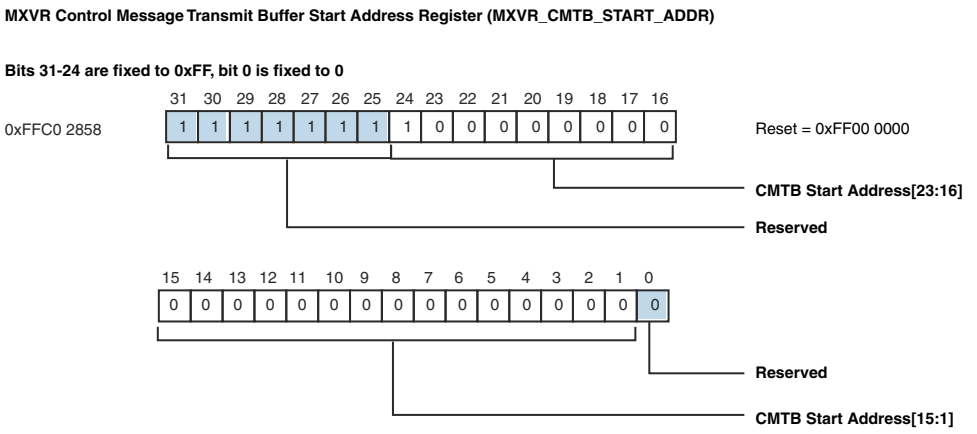


Figure 29-30. MXVR Control Message Transmit Buffer Start Address Registers

MXVR Control Message Transmit Buffer Current Address (MXVR_CMTB_CURR_ADDR) Register

This register provides the current address that the Control Message Transmit DMA channel is reads from in the `CMTB`. The `CMTB` can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 are always 1s and bit 0 are always 0.

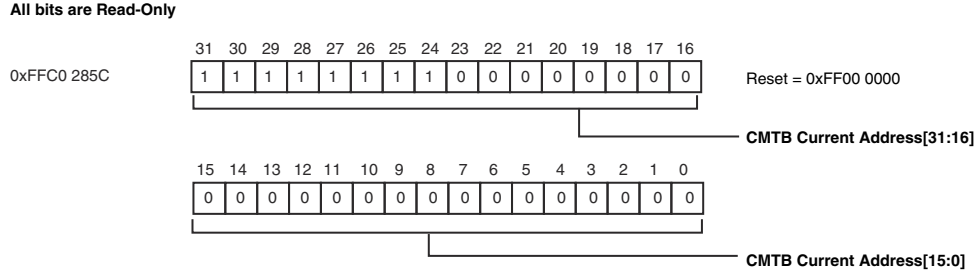


Figure 29-31. MXVR Control Message Transmit Buffer Current Address Register

MXVR Remote Read Buffer Start Address (MXVR_RRDB_START_ADDR) Register

The MXVR_RRDB_START_ADDR register sets the starting address for the Remote Read Buffer (RRDB) in L1 or L2 memory. The RRDB must be allocated 258 bytes. The RRDB can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 are fixed to 1s and bit 0 is fixed to 0.

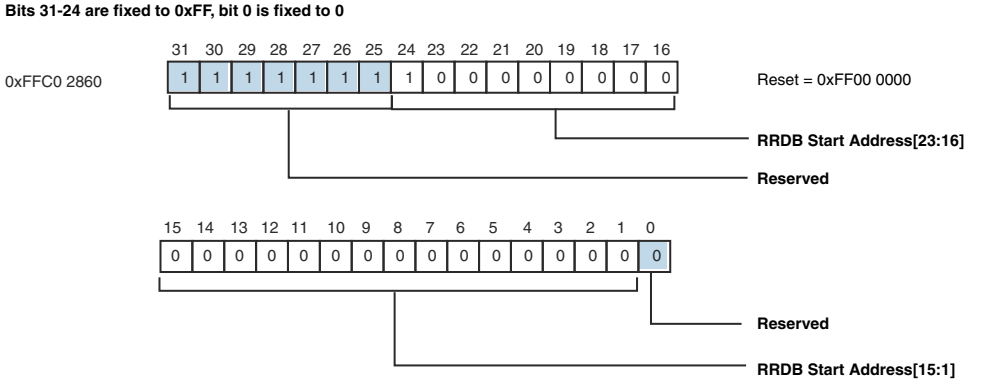


Figure 29-32. MXVR Remote Read Buffer Start Address Register

MXVR Remote Read Buffer Current Address (MXVR_RRDB_CURR_ADDR) Register

The MXVR_RRDB_CURR_ADDR register is a read-only register which gives the current address that the Remote Read Buffer DMA channel is reading or writing in the RRDB. The RRDB can only reside in L1 or L2 memory and must be word aligned. Therefore, bits 31–25 will always be 1s and bit 0 will always be 0.

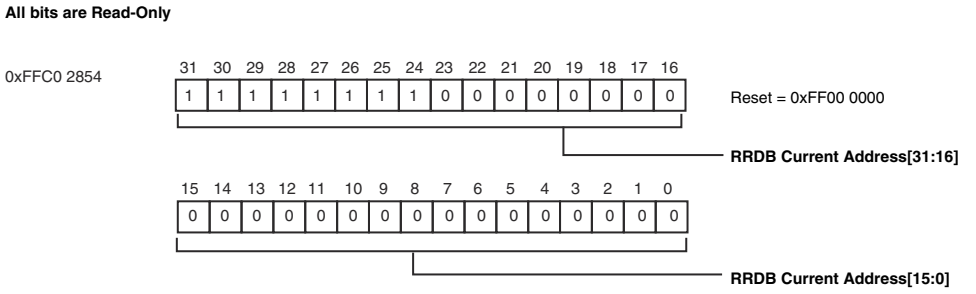


Figure 29-33. MXVR Remote Read Buffer Current Address Register

MXVR Pattern Registers

The MXVR has two sets of Pattern Registers: Pattern 0 Registers (PR0) and Pattern 1 Registers (PR1). Each set of Pattern Registers contains a data register and an enable register. The pattern matching registers define a pattern which a synchronous DMA channel will search for in the incoming datastream when the DMA channel is in one of the Synchronous Packet modes. The MXVR_DMAx_CONFIG registers allow the “start pattern” to be defined by either PR0 or PR1 and the “stop pattern” to be defined by either PR0 or PR1. The patterns can be from one to four bytes long and can be enabled in a bit-wise manner to allow “don't cares”.

MXVR Pattern Data (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1) Registers

The MXVR_PAT_DATA_x registers contain the data value to be used in the comparison with received synchronous data in a logical channel while checking for the occurrence of a “start pattern” or a “stop pattern” when a DMA channel is in one of the Synchronous Packet modes. The data register is four bytes long. Pattern matching will only be checked on byte boundaries and can match across frames within the same logical channel.

The programming of the MXVR_PAT_EN_x registers determines which of the bits in each of the four bytes will be used to check for a pattern match and controls the number of bytes to be matched.

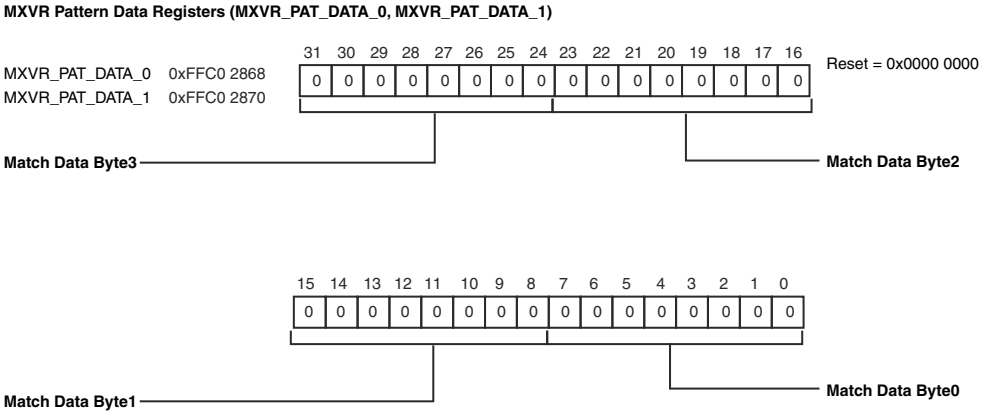


Figure 29-34. MXVR Pattern Data Registers

MXVR Pattern Enable (MXVR_PAT_EN_0, MXVR_PAT_EN_1) Registers

The MXVR_PAT_EN_x registers contain bit enables that allow individual bits within a Match Data Byte to be selectively enabled (a “care”) or disabled (a “don't care”) in the comparison with the incoming synchronous data bytes in a logical channel while checking for the occurrence of a “start pattern” or a “stop pattern” when a DMA channel is in one of the Synchronous Packet modes.

For example, if the Byte0-Bit 7 Match Enable is set to 1 and all the other bit match enables are set to 0, then only bit 7 of Match Data Byte 0 is used in the comparison and bits 6–0 of Match Data Byte 0 are “don't cares”. Therefore, for a pattern match to occur, only bit 7 of the received synchronous data byte in the logical channel must match bit 7 of Match Data Byte 0.

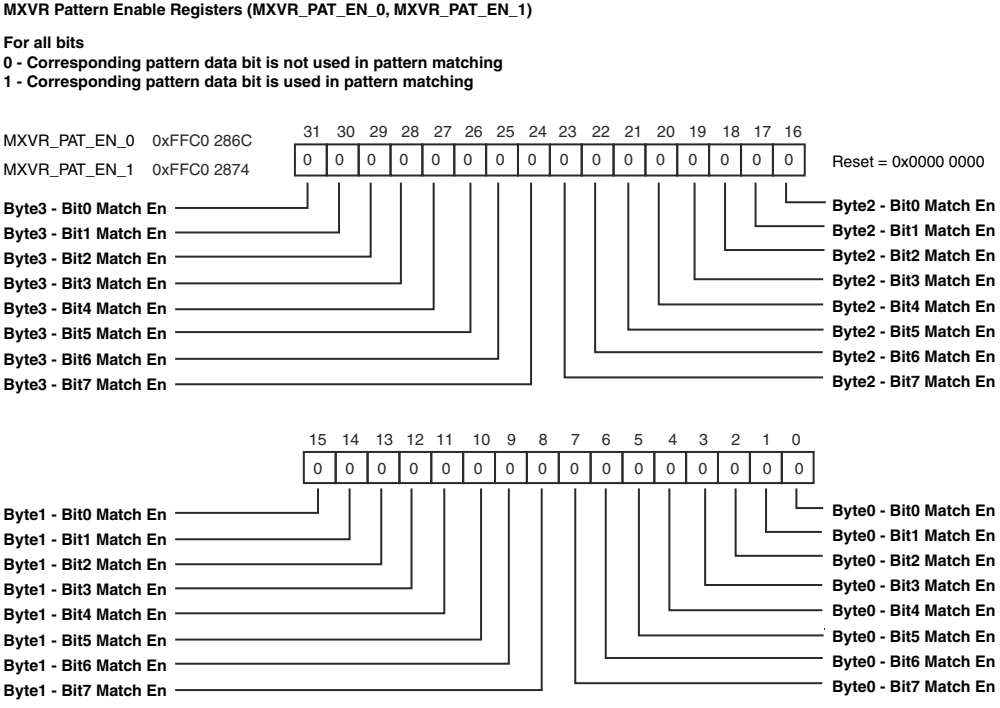


Figure 29-35. MXVR Pattern Enable Registers

The number of bytes to be used in pattern matching is also determined by the MXVR_PAT_EN_x. The number of bytes to be used in matching is determined by the highest byte number to have at least one bit enabled in the MXVR_PAT_EN_x. For example, if any bit in Data Byte 3 is enabled for matching then all 4 bytes will be used in pattern matching. If no bits are enabled in Data Byte 3, Data Byte 2 and Data Byte 1, and some bits are enabled in Data Byte 0 then only one byte (Data Byte 0) will be used in pattern matching.

MXVR Frame Counter (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1) Registers

The MXVR has two completely independent frame counters which each have an interrupt. Each frame counter is a down-counter which decrements when the MXVR is frame locked and whenever a preamble is received (at the beginning of every frame). The frame counter can optionally generate an interrupt when the counter reaches zero. The frame counter decrements on all types of preambles. The frame counter is controlled by accessing the MXVR_FRAME_CNT_x register. Writing the MXVR_FRAME_CNT_x register reloads the frame counter with the 16-bit value written and starts the counter decrementing when the MXVR is frame locked and a preamble is received. If the MXVR loses frame lock after the frame counter is started, the frame counter will pause until the MXVR is back in frame lock. The value written must be between 0x0001 and 0xFFFFF. Once the frame counter decrements to zero, the corresponding Frame Counter Zero (FCZ0 or FCZ1) bit in the MXVR_INT_STAT_0 register will change to 1 and the Status Change Interrupt will assert if the corresponding Frame Counter Zero Interrupt Enable (FCZ0EN, or FCZ1EN) bit in the MXVR_INT_EN_0 register is set to 1. The FCZ0 and FCZ1 bits in the MXVR_INT_STAT_0 register are sticky bits which must be written with a 1 in order to clear the bit and clear the interrupt. The frame counters can be stopped and reset at any time by writing 0x0000 to the MXVR_FRAME_CNT_x register and no interrupt will be generated. Reading the MXVR_FRAME_CNT_x will return the current value of the frame counter.

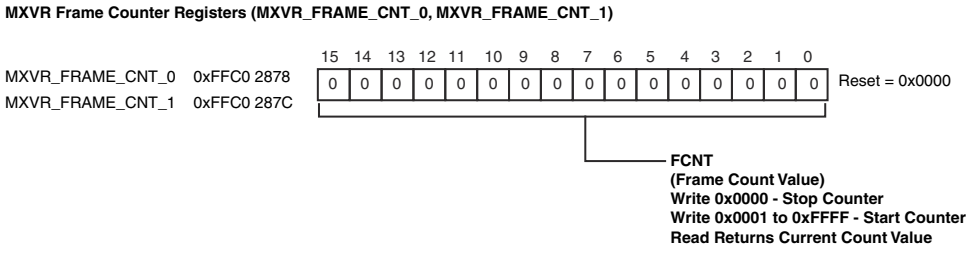


Figure 29-36. MXVR Frame Counter Registers

MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers

The MXVR_ROUTING_x registers are used to route data from one synchronous data channel to another or to mute particular synchronous channels. The MXVR can route synchronous data received on one physical channel so that it is transmitted on one or more other physical channel. In addition, the MXVR_ROUTING_x registers may be used to mute one or more transmitted physical channels. When a synchronous data channel is muted, the data transmitted on that channel will be 0x00.

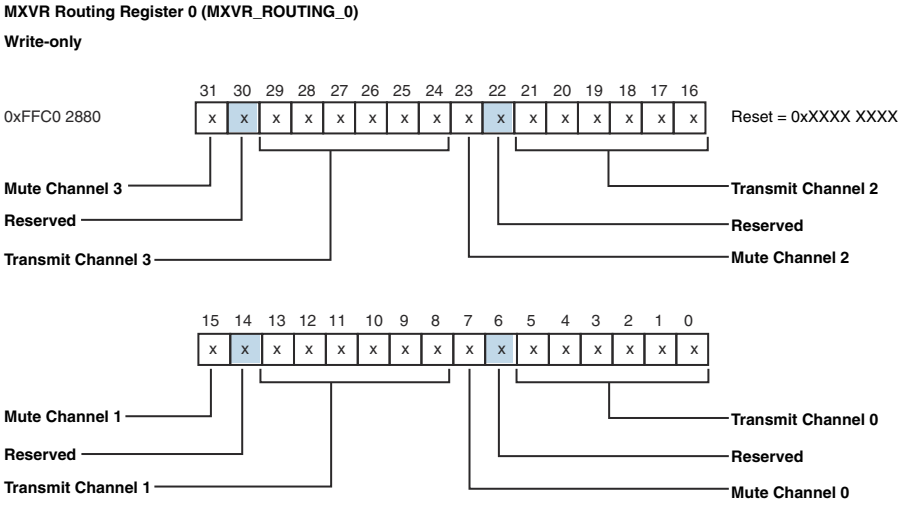


Figure 29-37. MXVR Routing Registers

All the Routing registers (MXVR_ROUTING_0 - MXVR_ROUTING_14) have the same register format but each contain routing and muting control for different channels. MXVR_ROUTING_0 contains channels 0 to 3, contains channels 4 to 7, and so on. Figure 29-37 shows MXVR_ROUTING_0 as an example of the register format.

The routing function can only be used for synchronous data channels (channel numbers less than 4 * RSB) when the MXVR is enabled and transmitting in Active Mode with a Synchronous Delay of two frames (when MXVREN = 1, MTXEN = 1, ACTIVE = 1 and SDELAY = 1 in the MXVR_CONFIG register). The muting function can be used for synchronous data channels when the MXVR is enabled and transmitting in Active Mode (MXVREN=1, MTXEN=1, and ACTIVE=1).

The MXVR_ROUTING_x registers are not reset, therefore they must be programmed to a known value after reset. In normal applications the data received on a particular physical channel should be routed to the same physical channel for transmission. Therefore, each Transmit Channel x entry will normally be programmed with the corresponding received channel number.

Synchronous data received on a particular physical channel can also be routed onto one or more different physical channels for transmission. For example, synchronous data received on physical channel 0 can be transmitted on physical channel 1 and synchronous data received on physical channel 1 can be transmitted on physical channel 0 by programming the Transmit Channel 0 to 0x01 and the Transmit Channel 1 to 0x00. The synchronous data received on a physical channel can also be transmitted on multiple channels. For example, synchronous data received on physical channel 5 can be transmitted on physical channels 8, 18, and 28 by programming Transmit Channel 8 to 0x05, Transmit Channel 18 to 0x05, and Transmit Channel 28 to 0x05.

In addition, the MXVR_ROUTING_x registers allow individual physical channels to be muted (causing the channel to transmit 0x00 regardless of what was received on that channel). When the Mute Channel x bit for a particular channel is set to 1, the channel will transmit 0x00 data regardless of the routing value programmed in the Transmit Channel x entry. In other words, the muting function takes precedence over the routing function.

The MXVR synchronous data DMA channels take precedence over the channel routing and channel muting functions. If a synchronous data DMA channel is enabled for transmit, the DMA'd data will be transmitted on the physical channels defined by the LCHAN field overriding any value programmed into the Transmit Channel entries or Mute Channel entries for those physical channels. When the DMA channel is disabled, however, the channel routing or channel muting function specified in the Transmit Channel entries and Mute Channel entries for those channels will be active. For example, if physical channels 4 and 5 have the Mute Channel x bit set, they will output 0x00 data. If a Logical Channel 0 is defined as physical channels 4 and 5 and a synchronous data DMA channel is setup to transmit on Logical Channel 0, once the DMA channel is enabled the DMA'ed data will be transmitted on physical channels 4 and 5. Once the synchronous data DMA channel is disabled, physical channels 4 and 5 will transmit 0x00 data again. This is particularly useful when transmitting synchronous packets in that the muting for the channels the synchronous packet is being sent on can be enabled so that before and after the synchronous packet data is set the synchronous channels will have all 0x00 data.

The MXVR_ROUTING_x registers are write-only. Reading any of the MXVR_ROUTING_x registers will result in a bus error exception and will return unknown data.

The routing and muting fields serve an additional purpose. The fields determine whether the MXVR will report a physical channel as being "In-Use". If the MXVR is muting a particular physical channel or if the MXVR is routing data from another channel onto that physical channel, the MXVR will report that physical channel is "In-Use" to the Master. If the MXVR is not muting a particular physical channel or is not routing data from another channel onto that physical channel, the MXVR will report that physical channel is not "In-Use". If The Master determines which channels are "In-Use" when the Allocation Table is distributed and the "Channel-In-Use" bits for all the nodes in the ring are available in the Master's MXVR_ALLOC_x registers.

MXVR Block Counter (MXVR_BLOCK_CNT) Register

The MXVR has a Block Counter which has an associated interrupt. The Block Counter is a down-counter which decrements when the MXVR is block locked and a normal block preamble is received and can optionally generate an interrupt when the counter reaches zero. The block counter does not decrement when the MXVR is not block locked or when the block preambles are received when the Allocation Table is being distributed over the control message channel. Two block preambles out of every sixty-four block preambles are for Allocation Table distribution blocks.

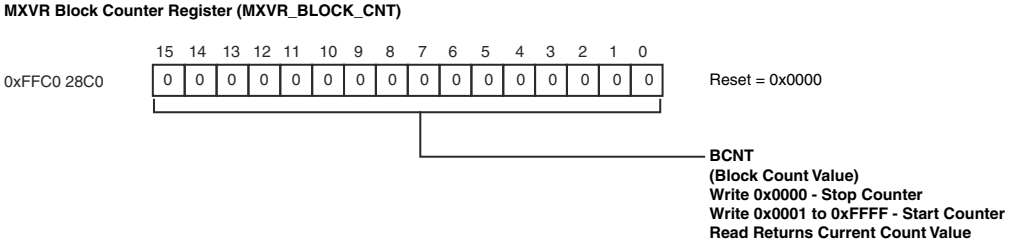


Figure 29-38. MXVR Block Counter Register

The block counter is controlled by accessing the MXVR_BLOCK_CNT register. Writing the MXVR_BLOCK_CNT register reloads the block counter with the 16-bit value written and starts the counter decrementing when the MXVR is block locked and a normal block preamble is received. The value written must be between 0x0001 and 0xFFFF. If the MXVR loses block lock after the block counter is started, the block counter pauses until the MXVR is back in block lock. Once the block counter decrements to zero, the Block Counter Zero (BCZ) bit in the MXVR_INT_STAT_0 register changes to 1 and the Status Change Interrupt asserts if the Block Counter Zero Interrupt Enable (BCZEN) bit in the MXVR_INT_EN_0 register is set to 1. The BCZ bit in the MXVR_INT_STAT_0 register is a sticky bit which must be written with a 1 in order to clear the bit and the interrupt. The block counter can be stopped and reset at any time by writing 0x0000 to the MXVR_BLOCK_CNT register and no interrupt will be generated. Reading the MXVR_BLOCK_CNT will return the current value of the block counter.

MXVR Clock Control (MXVR_CLK_CTL) Register

The MXVR_CLK_CTL register controls the MXVR Crystal Oscillator and the MXVR clock outputs.

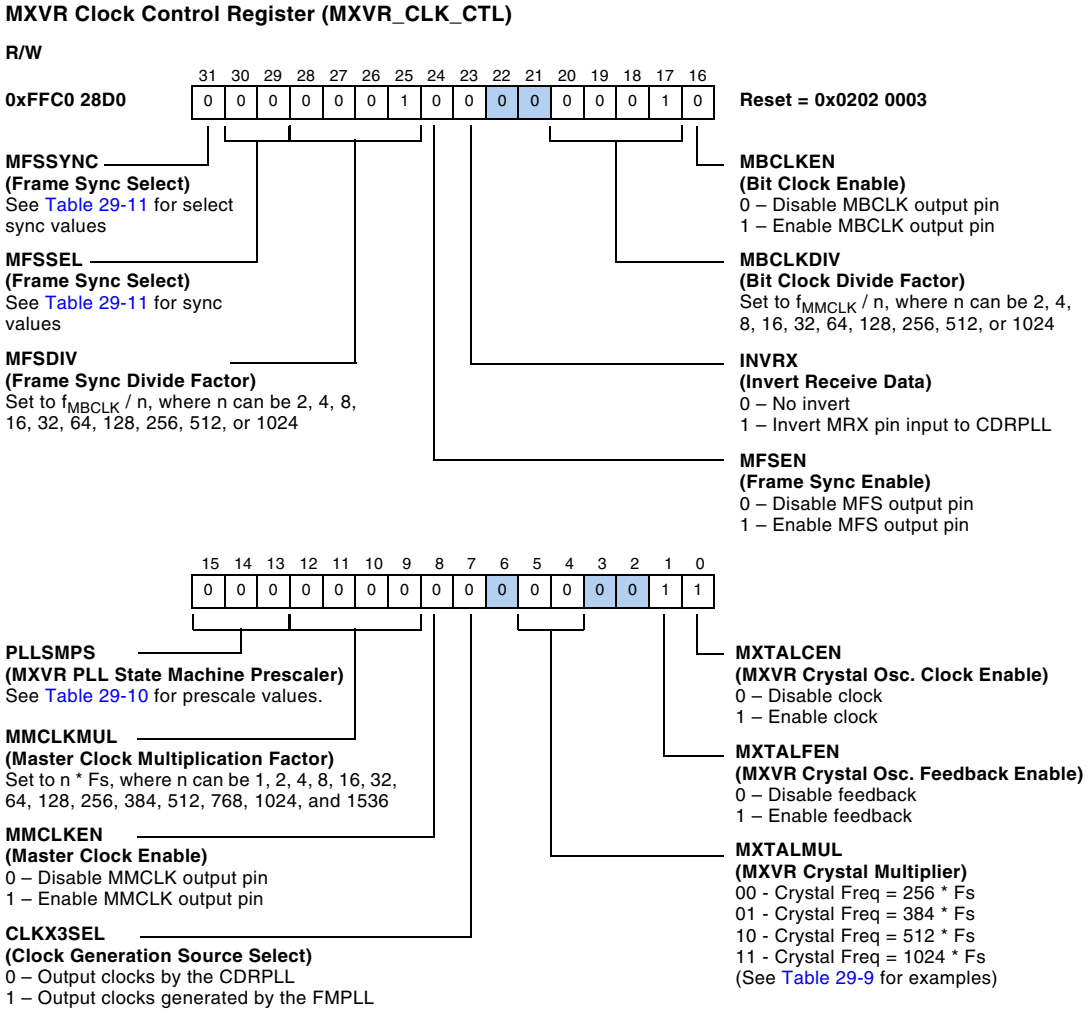


Figure 29-39. MXVR Clock Control Register

The MXVR Crystal Oscillator Clock Enable (MXTALCEN) bit enables or disables the clock output by the MXVR Crystal Oscillator which is used by the MXVR PLLs. The MXTALCEN bit enables or disables the clock regardless of whether a crystal is used between the MXI and MXO pins or whether a clock is directly driven into the MXI pin. When MXTALCEN is set to 1, the clock supplied by the MXVR Crystal Oscillator is enabled, and when MXTALCEN is set to 0, the clock supplied by the MXVR Crystal Oscillator is disabled. The MXTALCEN is set to 1 by reset. The MXTALCEN can be used to gate off the clock to the MXVR in order to save power when the network is not in operation. Note that the crystal should be at frequency and MXTALCEN should be enabled, or the clock driven on the MXI input should be at frequency and the MXTALCEN should be enabled prior to starting up the MXVR PLLs.

The MXVR Crystal Oscillator Feedback Enable (MXTALFEN) bit enables or disables the resistive feedback between the MXVR Crystal Input pin (MXI) and the MXVR Crystal Output pin (MXO). The MXVR Crystal Oscillator supplies a clock which is used by the MXVR PLLs. A crystal can be placed between the MXI and MXO pins (along with the appropriate capacitors) or a clock may be driven directly into the MXI pin and the MXO pin can be left unconnected. When using a crystal, if the MXTALFEN is set to 1, the resistive feedback between MXI and MXO is enabled and the crystal will oscillate. When using a crystal, if MXTALFEN is set to 0, the resistive feedback is disabled and the crystal will not oscillate. If a crystal is not used and a clock is driven directly onto the MXI pin, the MXTALFEN must be set to 1 for proper operation. The MXTALFEN is set to 1 by reset, so that if a crystal is used, the crystal will start up during the reset time and software boot time.

The MXVR must either be supplied with an externally generated clock driven on the MXI pin or must have a crystal (and appropriate external components) connected between the MXI and MXO pins. In either case, the frequency should be 256 * Fs, 384 * Fs, 512 * Fs, or 1024 * Fs. The frequency that is being supplied should be programmed into the MXTALMUL bits in the MXVR_CLK_CTL register. If a crystal is placed between MXI and MXO, and the network will be disabled for an extended period of time, the

MXTALFEN and the MXTALCEN can be set to 0 to decrease power consumption. If a clock is being directly driven to the MXI pin, and the network will be disabled for an extended period of time, the MXTALCEN can be set to 0 to decrease power consumption. However, the clock or crystal must be stable at frequency prior to starting up the MXVR PLLs in order to lock the network.

The MXVR Crystal Multiplier (MXTALMUL) field determines the multiplication factor that will be used when the MXVR PLLs are configured to multiply the crystal or input clock frequency up to the transmit clock frequency (1024 * Fs). Table 29-9 shows all the crystal frequencies (256 * Fs, 384 * Fs, 512 * Fs, or 1024 * Fs) which can be used to multiply up to the transmit clock frequency for sample frequencies of 38 kHz, 44.1 kHz, and 48 kHz.

Table 29-9. Crystal Input Frequencies

MXTALMUL	Multiply Factor	Crystal Frequency	Crystal Frequency Needed for Desired Fs		
			Fs = 38 kHz	Fs = 44.1 kHz	Fs = 48 kHz
b#00	8/2	256 * Fs	9.728 MHz	11.2896 MHz	12.288 MHz
b#01	8/3	384 * Fs	14.592 MHz	16.9344 MHz	16.432 MHz
b#10	8/4	512 * Fs	19.456 MHz	22.5792 MHz	24.576 MHz
b#11	8/8	1024 * Fs	38.912 MHz	45.1584 MHz	49.152 MHz

The Clock Generation Source Select (CLKX3SEL) bit selects whether the MMCLK, MBCLK, and MFS output clocks are generated by the FMPLL or by the CDRPLL. If the CLKX3SEL bit is set to 1, the FMPLL will generate the MMCLK, MBCLK, and MFS output clocks. If the CLKX3SEL bit is set to 0, the CDRPLL will generate the MMCLK, MBCLK, and MFS output clocks. The CLKX3SEL bit is set to 0 by reset.

The Master Clock Enable (MMCLKEN) bit enables or disables the MXVR Master Clock output pin (MMCLK). If the MMCLKEN bit is set to 0, the MMCLK pin will remain at a logic low level. If the MMCLKEN bit is set to 1, the MMCLK

pin will supply a clock at a frequency determined by the MMCLKMUL field. The MMCLKEN bit is set to 1 by reset. After reset is negated the MMCLK output pin will remain low and will not toggle until the MXVR PLL which is selected to generate the output clocks is started-up and the MMCLKEN bit is set to 1.

The Master Clock Multiplication Factor (MMCLKMUL) field determines the frequency of the MXVR Master Clock output pin (MMCLK). The MMCLK clock frequency can be specified as a multiplication of the sample rate (Fs). The frequency can be set to be n * Fs where n can be 1, 2, 4, 8, 16, 32, 64, 128, 256, 384, 512, 768, 1024, and 1536. When MMCLKMUL is set to any value except 1024 * Fs or 1536 * Fs, the MMCLK duty cycle will be 50%. When MMCLKMUL is set to 1024 * Fs or 1536 * Fs, the MMCLK duty cycle will be 33%. Note that the MMCLKMUL should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to 0.

The MXVR PLL State Machine Prescaler (PLLSMPS) field is a prescale value used by the FMPLL and CDRPLL lock counters in order to adjust the lock times based on the SCLK frequency. Table 29-10 shows how the PLLSMPS field should be programmed based on the SCLK frequency.

Table 29-10. PLLSMPS Encoding Selection

SCLK Frequency Range	PLLSMPS
116MHz < f_{SCLK} <= 133MHz	b#000
99MHz < f_{SCLK} <= 116MHz	b#001
83MHz < f_{SCLK} <= 99MHz	b#010
66MHz < f_{SCLK} <= 83MHz	b#011
49MHz < f_{SCLK} <= 66MHz	b#100
33MHz < f_{SCLK} <= 49MHz	b#101
16MHz < f_{SCLK} <= 33MHz	b#110
f_{SCLK} <= 16MHz	b#111

The Bit Clock Enable (MBCLKEN) bit enables or disables the MXVR Bit Clock output pin (MBCLK). If the MBCLKEN bit is set to 0, the MBCLK pin will remain at a logic low level. If the MBCLKEN bit is set to 1, the MBCLK pin will supply a clock at a frequency determined by the MBCLKDIV field. MBCLKEN is set to 0 by reset. After reset is negated, the MBCLK output pin will remain low and will not toggle until the MXVR PLL which is selected to generate the output clocks is started-up and the MBCLKEN bit is set to 1.

The Bit Clock Divide Factor (MBCLKDIV) field determines the frequency of the MXVR Bit Clock output pin (MBCLK). The clock output on the MBCLK pin is generated by the MXVR Master Clock. The MBCLK clock frequency can be specified as a division of the MXVR Master Clock frequency. The frequency can be set to be f_{MMCLK} / n where n can be 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024. When MMCLKMUL is set to any value except 1024 * Fs or 1536 * Fs, the rising edge of MBCLK occurs in sync with the rising edge of MMCLK. When MMCLKMUL is set to 1024 * Fs or 1536 * Fs, the rising edge of MBCLK occurs in sync with the falling edge of MMCLK. Note that the MBCLKDIV should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to 0.

The Invert Receive (INVRX) bit determines whether the incoming data stream on the MXVR Receive Data input pin (MRX) will feed into the CDRPLL as is or whether the data stream will be inverted before feeding into the CDRPLL. If the INVRX bit is set to 0, the data stream will feed into the CDRPLL as is. If the INVRX bit is set to a 1, the data stream will be inverted prior to being fed into the CDRPLL.

The Frame Sync Enable (MFSEN) bit enables or disables the MXVR Frame Sync output pin (MFS). If the MFSEN bit is set to 0 and MFSSEL is set to b#00 (Clock Mode), the MFS pin will remain at a logic low level. If the MFSEN bit is set to 0 and MFSSEL is set to b#01 (Active-High Pulse Mode), the MFS pin will remain at a logic low level. If the MFSEN bit is set to 0 and MFSSEL is set to b#10 (Active-Low Pulse Mode), the MFS pin will remain at a logic high level. If the MFSEN bit is set to 1, the MFS pin will supply a clock or pulse at a frequency determined by the MFSDIV field. MFSEN is set to 0 by

reset. After reset is negated the MFS output pin will remain in its inactive state and will not toggle until the MXVR PLL which is selected to generate the output clocks is started-up and the MFSEN bit is set to 1.

The Frame Sync Divide Factor (MFSDIV) field determines the frequency of the MXVR Frame Sync output pin (MFS). The clock output on the MFS pin is generated by the MXVR Bit Clock. The MFS clock frequency can be specified as a division of the MXVR Bit Clock frequency. The frequency can be set to be f_{MBCLK} / n where n can be 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024. Note that the MFSDIV should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to 0.

The Frame Sync Select (MFSSEL) field determines whether the MXVR Frame Sync output pin (MFS) will generate a 50% duty cycle clock, an active-high pulse, or an active-low pulse. If the MFSSEL field is set to b#00, the MFS will generate a 50% duty cycle clock. If the MFSSEL field is set to b#01, the MFS will generate an active-high pulse with a pulse length equal to the MBCLK period. If the MFSSEL field is set to 10, the MFS will generate an active-low pulse with a pulse length equal to the MBCLK period. Note that the MFSSEL should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to 0.

The Frame Sync Synchronization Select (MFSSYNC) bit determines the synchronization between the MFS and MBCLK output pins. If the MFS is programmed to be a 50% duty cycle clock (MFSSEL = b#00) or an active-high pulse (MFSSEL = 01) and the MFSSYNC is set to 0, the rising edge of MFS occurs in sync with the falling edge of the MBCLK. If the MFS is programmed to be a 50% duty cycle clock (MCLKSEL = b#00) or an active-high pulse (MCLKSEL = b#01) and the MFSSYNC is set to 1, the rising edge of MFS occurs in sync with the rising edge of the MBCLK. If MFS is programmed to be a active-low pulse (MFSSEL = b#10) and the MFSSYNC is set to 0, the falling edge of MFS occurs in sync with the falling edge of MBCLK. If MFS is programmed to be a active-low pulse (MFSSEL = b#10) and the MFSSYNC is set to 1, the falling edge of MFS occurs in sync with the rising edge of MBCLK.

Table 29-11. Frame Sync Synchronization (MFSSYNC) Selections

MFSSYNC	MFSSEL	Frame Sync Synchronization
0	b#00	The rising edge of MFS occurs in sync with the falling edge of the MBCLK
	b#01	The rising edge of MFS occurs in sync with the falling edge of the MBCLK
	b#10	The falling edge of MFS occurs in sync with the falling edge of MBCLK
1	b#00	The rising edge of MFS occurs in sync with the rising edge of the MBCLK
	b#01	The rising edge of MFS occurs in sync with the rising edge of the MBCLK
	b#10	The falling edge of MFS occurs in sync with the rising edge of MBCLK

MXVR Clock/Data Recovery PLL Control (MXVR_CDRPLL_CTL) Register

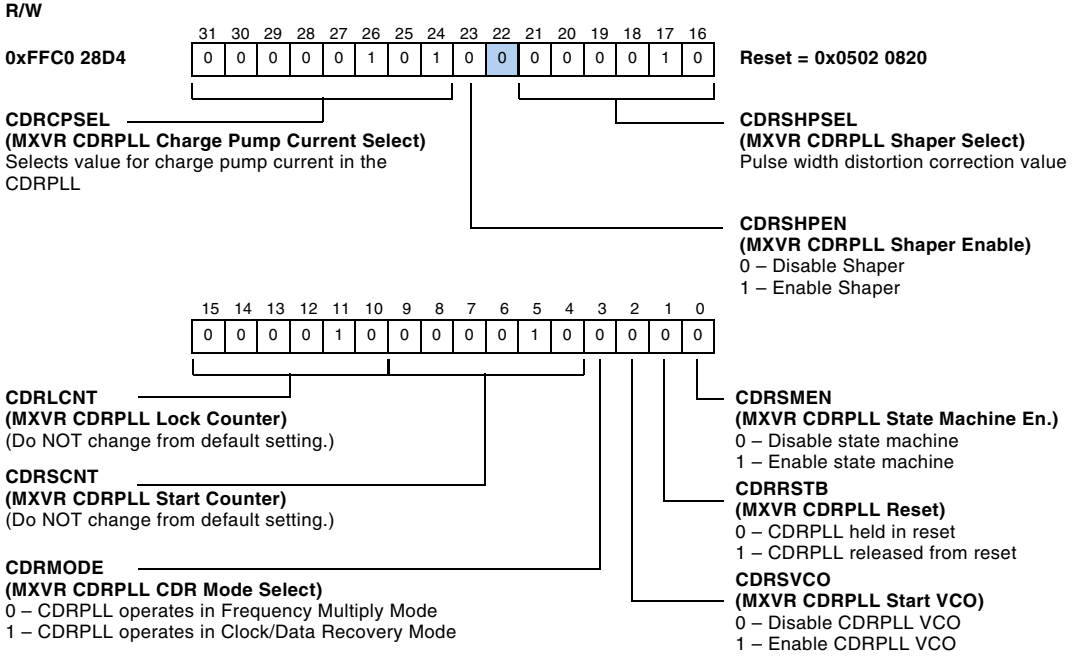


Figure 29-40. MXVR Clock/Data Recovery PLL Control Register

The MXVR Clock/Data Recovery PLL State Machine Enable (CDRSMEN) bit enables or disables the state machine which controls the CDRPLL. When CDRSMEN bit is set to a 1, the CDRPLL state machine will start up the CDRPLL and control its operation. When the CDRSMEN bit is set to a 0, the CDRPLL state machine is disabled and the CDRRSTB, CDRSVCO, and CDRMODE bits directly control the operation of the CDRPLL. It is recommended that the CDRPLL state machine be used to control the

CDRPLL rather than directly controlling the CDRRSTB, CDRSVCO, and CDRMODE bits. The CDRSMEN bit is set to 0 by reset.

The MXVR Clock/Data Recovery PLL Reset (CDRRSTB) bit controls the reset to the CDRPLL if the CDRPLL state machine is disabled. When the CDRPLL state machine is disabled and the CDRRSTB bit is set to a 0, the CDRPLL is held in reset. When the CDRPLL state machine is disabled and the CDRRSTB bit is set to a 1, the CDRPLL is released from reset. When the CDRPLL state machine is enabled, the CDRPLL state machine controls the CDRPLL and therefore the CDRRSTB bit has no effect. It is recommended that the CDRPLL state machine be used to control the CDRPLL rather than directly controlling the CDRRSTB bit. The CDRRSTB bit is set to 0 by reset.

The MXVR Clock/Data Recovery PLL Start VCO (CDRSVCO) bit controls the startup of the VCO in the CDRPLL if the CDRPLL state machine is disabled. When the CDRPLL state machine is disabled and the CDRSVO bit is set to a 0, the CDRPLL VCO is disabled. When the CDRPLL state machine is disabled and the CDRSVCO bit is set to a 1, the CDRPLL VCO is enabled. When the CDRPLL state machine is enabled, the CDRPLL state machine controls the CDRPLL and therefore the CDRSVCO bit has no effect. It is recommended that the CDRPLL state machine be used to control the CDRPLL rather than directly controlling the CDRSVCO bit. The CDRSVCO bit is set to 0 by reset.

The MXVR Clock/Data Recovery PLL CDR Mode Select (CDRMODE) bit controls whether the CDRPLL is in Frequency Multiply Mode or Clock/Data Recovery Mode if the CDRPLL state machine is disabled.

When the CDRPLL state machine is disabled and the CDRMODE bit is set to a 0, the CDRPLL operates in Frequency Multiply Mode. When the CDRPLL state machine is disabled and the CDRMODE bit is set to a 1, the CDRPLL operates in Clock/Data Recovery Mode. When the CDRPLL state machine is enabled, the CDRPLL state machine controls the CDRPLL and therefore the CDRMODE bit has no effect. It is recommended that the CDRPLL state machine be used to control the CDRPLL rather than directly controlling the CDRMODE bit. The CDRMODE bit is set to 0 by reset.

The MXVR Clock/Data Recovery PLL Start Counter (CDRSCNT) field controls the start-up time of the CDRPLL if the CDRPLL state machine is enabled. The CDRSCNT field is set to b#000010 by reset. It is recommended that the CDRSCNT field not be changed from its reset value.

The MXVR Clock/Data Recovery PLL Lock Counter (CDRLCNT) field controls the lock time of the CDRPLL if the CDRPLL state machine is enabled. The CDRLCNT field is set to b#000010 by reset. It is recommended that the CDRLCNT field not be changed from its reset value.

The MXVR Clock/Data Recovery PLL Shaper Select (CDRSHPSEL) field controls the amount of pulse width distortion correction to be made to the incoming data stream when the CDRPLL Shaper is enabled. The CDRSHPSEL field is set to b#000000 by reset.

The MXVR Clock/Data Recovery PLL Shaper Enable (CDRSHPEN) bit enables or disables the CDRPLL Shaper which corrects pulse width distortion in the incoming data stream. The CDRSHPSEL bit is set to 0 by reset.

The MXVR Clock/Data Recovery PLL Charge Pump Current Select (CDRCPSEL) field controls the charge pump current in the CDRPLL. The CDRCPSEL field is set to 0x05 by reset.

MXVR Frequency Multiply PLL Control (MXVR_FMPLL_CTL) Register

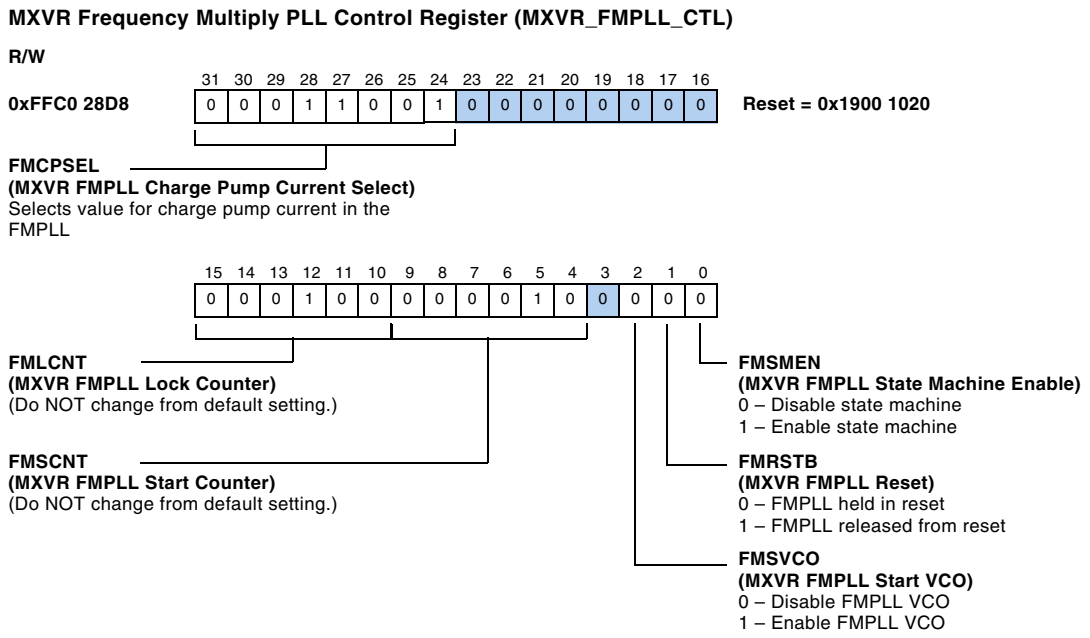


Figure 29-41. MXVR Frequency Multiply PLL Control Register

The MXVR Frequency Multiply PLL State Machine Enable (FMSMEN) bit enables or disables the state machine which controls the FMPLL. When FMSMEN bit is set to a 1, the FMPLL state machine will start up the FMPLL and control its operation. When the FMSMEN bit is set to a 0, the FMPLL state machine is disabled and the FMRSTB and FMSVCO bits directly control the operation of the FMPLL. It is recommended that the FMPLL state machine be used to control the FMPLL rather than directly controlling the FMRSTB and FMSVCO bits. The FMSMEN bit is set to 0 by reset.

The MXVR Frequency Multiply PLL Reset (FMRSTB) bit controls the reset to the FMPLL if the FMPLL state machine is disabled. When the FMPLL state machine is disabled and the FMRSTB bit is set to a 0, the FMPLL is

held in reset. When the FMPLL state machine is disabled and the FMRSTB bit is set to a 1, the FMPLL is released from reset. When the FMPLL state machine is enabled, the FMPLL state machine controls the FMPLL and therefore the FMRSTB bit has no effect. It is recommended that the FMPLL state machine be used to control the FMPLL rather than directly controlling the FMRSTB bit. The FMRSTB bit is set to 0 by reset.

The MXVR Frequency Multiply PLL Start VCO (FMSVCO) bit controls the startup of the VCO in the FMPLL if the FMPLL state machine is disabled. When the FMPLL state machine is disabled and the FMSVO bit is set to a 0, the FMPLL VCO is disabled. When the FMPLL state machine is disabled and the FMSVCO bit is set to a 1, the FMPLL VCO is enabled. When the FMPLL state machine is enabled, the FMPLL state machine controls the FMPLL and therefore the FMSVCO bit has no effect. It is recommended that the FMPLL state machine be used to control the FMPLL rather than directly controlling the FMSVCO bit. The FMSVCO bit is set to 0 by reset.

The MXVR Frequency Multiply PLL Start Counter (FMSCNT) field controls the start-up time of the FMPLL if the FMPLL state machine is enabled. The FMSCNT field is set to b#000001 by reset. It is recommended that the FMSCNT field not be changed from its reset value.

The MXVR Frequency Multiply PLL Lock Counter (FMLCNT) field controls the lock time of the FMPLL if the FMPLL state machine is enabled. The FMLCNT field is set to b#000100 by reset. It is recommended that the FMLCNT field not be changed from its reset value.

The MXVR Frequency Multiply PLL Charge Pump Current Select (FMCPSEL) field controls the charge pump current in the FMPLL. The FMCPSEL field is set to 0x19 by reset.

MXVR Pin Control (MXVR_PIN_CTL) Register

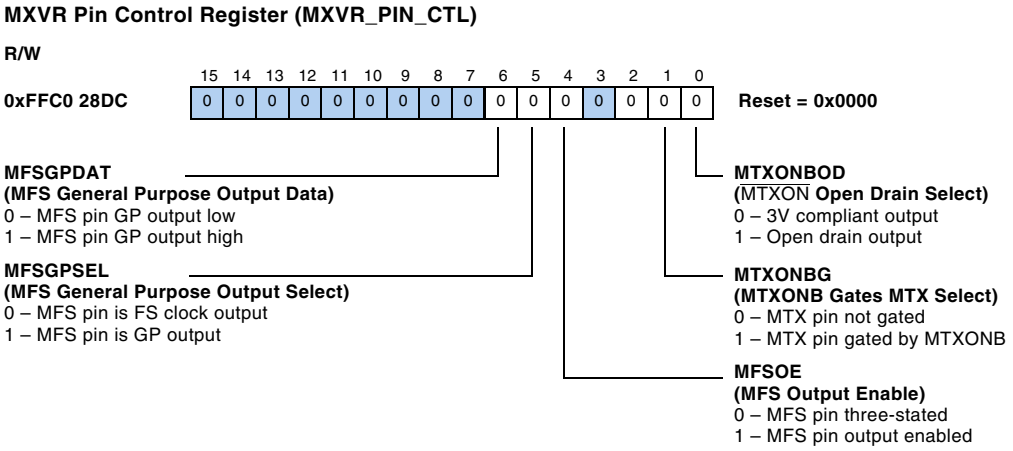


Figure 29-42. MXVR Pin Control Register

The $\overline{\text{MTXON}}$ Open Drain Select (MTXONBOD) bit controls whether the $\overline{\text{MTXON}}$ pin operates as a 3V compliant output or as a 5V tolerant open drain output. Normally, the $\overline{\text{MTXON}}$ pin is connected to a transistor to turn on and off the power supply to the Transmit PHY. If the Transmit PHY has a 3V power supply, the MTXONBOD can be set so that the $\overline{\text{MTXON}}$ pin operates as a 3V compliant output and can be connected to the transistor. If the Transmit PHY has a 5V power supply, the MTXONBOD can be set so that the $\overline{\text{MTXON}}$ pin operates as a 5V tolerant open drain output and can be connected to the transistor with a pull-up resistor to 5V. When the MTXONBOD is set to 0, the $\overline{\text{MTXON}}$ pin operates as a 3V compliant output. When the MTXONBOD is set to 1, the $\overline{\text{MTXON}}$ pin operates as a 5V tolerant open drain output. The MTXONBOD bit is set to 0 by reset.

The MTXONB Gates MTX Select (MTXONBG) bit controls whether the MTX pin is gated based on the state of the MTXONB bit in the MXVR_CONFIG register. When the MTXONBG bit is set to 0, the MTX pin can toggle regardless of the state of the MTXONB bit in the MXVR_CONFIG register. When the MTXONBG bit is set to 1 and the MTXONB bit in the MXVR_CONFIG register is set to 1, the

MTX pin will be driven to a logic low level. When the MTXONBG bit is set to 1 and the MTXONB bit in the MXVR_CONFIG register is 0, the MTX pin is allowed to toggle. Gating the MTX pin with the MTXONB bit may be desirable since some Transmit PHYs may partially power up when the MTX pin toggles even when the power is turned off to the Transmit PHY. The MTXONBG bit is set to 0 by reset.

The MFS Pin Output Enable (MFSOE) bit controls whether the MFS output pin is three-stated or output enabled. When the MFSOE bit is set to 0, the MFS output pin will three-state. When the MFSOE bit is set to 1, the MFS output pin will be output enabled. The MFSOE bit is set to 0 by reset.

The MFS Pin General Purpose Output Select (MFSGPSEL) bit controls whether the MFS pin outputs the MXVR Frame Sync clock output or acts as a general purpose output. When the MFSGPSEL bit is set to 0, the MFS pin will output the MXVR Frame Sync clock output. When the MFSGPSEL bit is set to 1, the MFS pin will act as a general purpose output pin controlled by the MFSGPDAT bit. The MFSGPSEL bit is set to 0 by reset.

The MFS Pin General Purpose Output Data (MFSGPDAT) bit controls the logic state of the MFS pin when the MFS is acting as a general purpose output. When the MFSGPSEL bit is set to 1 and the MFSGPDAT bit is set to 0, the MFS pin will output a logic low level. When the MFSGPSEL bit is set to 1 and the MFSGPDAT bit is set to 1, the MFS pin will output a logic high level. When the MFSGPSEL bit is set to 0, the state of the MFSGPDAT bit has no effect on the MFS pin. The MFSGPDAT bit is set to 0 by reset.

MXVR System Clock Counter (MXVR_SCLK_CNT) Register

The MXVR has a System Clock Counter which has an associated interrupt. The System Clock Counter is a down-counter which decrements once every 64 SCLK cycles and can optionally generate an interrupt when

the counter reaches zero. The System Clock Counter decrements regardless of the state of the MXVR (for example, regardless of whether the MXVR is enabled or disabled).

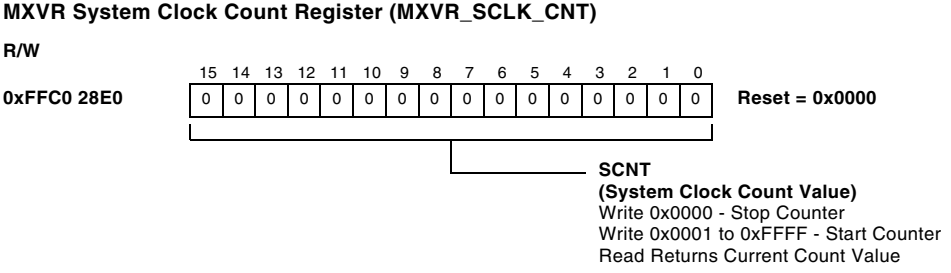


Figure 29-43. MXVR System Clock Count Register

The System Clock Counter is controlled by accessing the MXVR_SCLK_CNT register. Writing the MXVR_SCLK_CNT register reloads the System Clock Counter with the 16-bit value written and starts the counter decrementing. The value written must be between 0x0001 and 0xFFFF. Once the System Clock Counter decrements to zero, the System Clock Counter Zero (SCZ) bit in the MXVR_INT_STAT_0 register will change to 1 and the Status Change Interrupt will assert if the System Clock Counter Zero Interrupt Enable (SCZEN) bit in the MXVR_INT_EN_0 register is set to 1. The SCZ bit in the MXVR_INT_STAT_0 register is a sticky bit which must be written with a 1 in order to clear the bit and clear the interrupt. The System Clock Counter can be stopped and reset at any time by writing 0x0000 to the MXVR_SCLK_CNT register and no interrupt will be generated. Reading the MXVR_SCLK_CNT will return the current value of the System Clock Counter.

General Operation

The following sections describe MXVR general operations.

Network Services Software

Network Services Layer 1 and Layer 2 software is developed for the MXVR on the ADSP-BF54x processor which meets the MOST® Core Compliance specification. It is recommended that this software be used if the MXVR is to be operated in a MOST® compliant network. Contact Analog Devices for more information on the Network Services software stack.

Network Activity Detection

Network activity detection is done to indicate whether a node is receiving an active data stream. Typically an ADSP-BF54x processor MXVR master node is triggered to start up the network based on an external event (for example, car ignition, power switch, etc.), while an ADSP-BF54x processor MXVR slave node would normally operate in a low-power state until there is incoming network activity. Once incoming network activity is detected by the slave node, the MXVR will be started up, the Transmit PHY will be turned on, and the MXVR slave node will lock onto the incoming data stream. Once incoming network activity (circling the ring network) is detected by the master node, the MXVR master will lock onto the incoming data stream.

The MXVR has three methods for detecting network activity. One method monitors the state of the $\overline{\text{MRXON}}$ input, a second method detects edges on the MRX input, and a third method assumes that when ADSP-BF54x processor is powered-on that there is network activity. Note that the first two network activity detection methods can be utilized to generate interrupts even when the MXVR is disabled

The first method can be used in the case where the active-low status output of the Receive PHY is connected to the MXVR $\overline{\text{MRXON}}$ input. When the Receive PHY detects no network activity, the status output is high and when the Receive PHY detects network activity, the status output is low.

When the Receive PHY first detects network activity, the $\overline{\text{MRXON}}$ input will transition from high to low. The high to low transition on the $\overline{\text{MRXON}}$ input can wake the ADSP-BF54x processor from the hibernate state if the MXVRWE bit in the VR_CTL register is set to 1. For more information see Chapter 18, "Dynamic Power Management". A high to low transition on the $\overline{\text{MRXON}}$ input will set the MH2L bit in the MXVR_INT_STAT_0 register to 1 and if the MH2LEN bit in the MXVR_INT_EN_0 register is set to 1, an MXVR Status interrupt will be generated. The MXVR Status interrupt can also be programmed in the SIC_IWR1 register to wake the core from the Idle state.

When the Receive PHY detects a cessation of network activity, the $\overline{\text{MRXON}}$ input will transition from low to high. The low to high transition on the $\overline{\text{MRXON}}$ input will set the ML2H bit in the MXVR_INT_STAT_0 register to 1 and if the ML2HEN bit in the MXVR_INT_EN_0 register is set to 1, an MXVR Status interrupt will be generated. This interrupt on the cessation of network activity could be used to trigger the ADSP-BF54x processor to enter a low-power state.

In the second method for detecting network activity the MXVR detects edges on the MRX input. If a single rising or falling edge is detected on the MRX input, the MXVR will set the NACT bit in the MXVR_STATE_0 register to 1 indicating that there is network activity. If there are no rising or falling edges detected on the MRX input for 40 SCLK cycles, the MXVR will set the NACT bit to 0 indicating there is no network activity.

When the MXVR first detects network activity, the NACT bit will transition from low to high. The low to high transition of the NACT bit will set the NI2A bit in the MXVR_INT_STAT_0 register to 1 and if the NI2AEN bit in the MXVR_INT_EN_0 register is set to 1, an MXVR Status interrupt will be generated. The MXVR Status interrupt can also be programmed in the SIC_IWR1 to wake the core from the Idle state.

When the MXVR detects a cessation of network activity, the NACT bit will transition from high to low. The high to low transition of the NACT bit will set the NA2I bit in the MXVR_INT_STAT_0 register to 1 and if the NA2IEN bit

in the MXVR_INT_EN_0 register is set to 1, an MXVR Status interrupt will be generated. This interrupt on the cessation of network activity could be used to trigger the ADSP-BF54x processor to enter a low-power state.

The third method for network activity detection is handled completely outside of the ADSP-BF54x processor. In this method the Receive PHY status output controls the power supply for the ADSP-BF54x processor. When the Receive PHY status output indicates that there is no network activity, the power supply for the ADSP-BF54x processor is gated off. When the Receive PHY status output indicates that there is network activity, the power supply for the ADSP-BF54x processor is turned on. Once the reset to the ADSP-BF54x processor negates after the power-on-reset, the software can assume that there is network activity.

Node Initialization

Prior to starting up the MXVR PLL and enabling the MXVR ADSP-BF54x processor pin multiplexing, the MXVR_CONFIG register, the MXVR_CLK_CTL register, the MXVR_ROUTING_x registers, and the buffer start address registers must be initialized. The initialization of the MXVR_CONFIG register differs between a node to be started up in Master mode and a node to be started up in Slave mode. The initialization of the MXVR_CLK_CTL and the MXVR_ROUTING_x registers is the same for Master and Slave mode.

Initialization of Processor Pin Multiplexing

The GPIO_x_FER and GPIO_x_MUX registers for ports C, H, and G must be programmed to select the MXVR pins. For more information on how the GPIO registers should be programmed see Chapter 9, “General-Purpose Ports”.

Master Mode Initialization of MXVR_CONFIG Register

The MXVREN bit should remain 0 (keeping the MXVR disabled) until the MXVR PLLs are started up. The MMSM bit should be set to 1, the ACTIVE bit should be set to 1, the SDELAY bit should be set to 1, the NCMRXEN should be set to 0, and the RWRRXEN should be set to 0. The MTXEN bit should be set to 0 and the MTXONB bit should be set to 1 to keep the Transmit PHY turned off until the MXVR is enabled. The EPARITY bit should normally be set to 1 to select Even Parity. The MSB field should be set to a value less than b#0110 to indicate the ring is not yet locked. The APRXEN should be set to 0. The LMECH bit should be set to 0 or 1 depending on the desired locking mechanism.

Slave Mode Initialization of MXVR_CONFIG Register

The MXVREN bit should remain 0 (keeping the MXVR disabled). The MMSM bit should be set to 0, the ACTIVE bit should be set to 1, the SDELAY bit can be set to either 0 or 1, the NCMRXEN should be set to 0, and the RWRRXEN should be set to 0. The MTXEN bit should be set to 0 and the MTXONB bit should be set to 1 to keep the Transmit PHY turned off until the MXVR is enabled. The EPARITY bit should normally be set to 1 to select Even Parity. The MSB field is a don't care in slave mode. The APRXEN should be set to 0 and the WAKEUP bit should be set to 0. The LMECH bit is a don't care in slave mode.

Initialization of the MXVR_CLK_CTL Register

The MXTALCEN and MXTALFEN bits are both reset to 1 to allow a crystal connected between MXI and MXO to start-up immediately following the negation of reset. If either or both of these bits were set to 0 to save power, they must be set to 1 prior to starting up the MXVR PLLs. If a crystal is used, enough time should be allowed for the crystal to start-up prior to enabling the MXVR PLLs. The MXTALMUL bits should be set based on the

frequency of the crystal or clock driven into MXI. The MMCLKEN, MBCLKEN, and MFSEN bits should be set to 0. The state of the other bits in the MXVR_CLK_CTL register do not matter until the MXVR PLLs are started up.

Initialization of the MXVR_ROUTING_x Registers

Unless specific rerouting of synchronous data between received and transmitted physical channels is desired once the MXVR is enabled and activated, the Transmit Channel x fields should be written to forward each received channel to the corresponding transmitted channel. In addition, unless specific channel muting is desired, the Mute Channel x fields should be programmed to disable muting. For example, the MXVR_ROUTING_x registers could be written to forward all channels and disable all muting as follows:

```
*pMXVR_ROUTING_0 = 0x0302 0100;

*pMXVR_ROUTING_1 = 0x0706 0504;

*pMXVR_ROUTING_2 = 0x0B0A 0908;

…

*pMXVR_ROUTING_13 = 0x3736 3534;

*pMXVR_ROUTING_14 = 0x3B3A 3938;
```

Initialization of the Buffer Start Address Registers

The control message transmit and receive buffers, the asynchronous packet transmit and receive buffers, and the remote read buffer should be allocated space in L1 or L2 memory. The starting address of these buffers should then be programmed into the MXVR_CMTB_START_ADDR, MXVR_CMRB_START_ADDR, MXVR_APTB_START_ADDR, MXVR_APRB_START_ADDR, and MXVR_RRDB_START_ADDR registers.

Start Up of the MXVR PLLs

The initialization of the MXVR PLLs differs between a node to be started up in Master mode and a node to be started up in Slave mode. In a Master node, both the FMPLL and CDRPLL are used while in a Slave node only the CDRPLL is used.

Master Mode Initialization and Start Up of MXVR FMPLL and CDRPLL

Once the clock supplied on the MXI is stable and at frequency, the MXVR FMPLL and CDRPLL can be started up. Prior to starting up the PLLs, be sure that the MPS field in the MXVR_CLK_CTL register is programmed properly based on the SCLK frequency as this prescaler is used to control the PLL state machine lock times.

The FMPLL state machine will control the FMPLL start up and lock sequence. In order to enable the FMPLL state machine, the MXVR_FMPLL_CTL register should normally be programmed with the FMSMEN bit set to 1 and with all other fields set to their reset values. Once the FMPLL has started up, has locked to the MXI clock and is operating at the network frequency 1024 * Fs, the PFL bit in the MXVR_INT_STAT_0 register will be set to 1 and can optionally be enabled to generate a Status Change Interrupt. The current state of the FMPLL state machine can also be monitored by looking at the FMPLLST field in the MXVR_STATE_0 register. At the point when the PFL bit is asserted due to FMPLL locking at the network frequency, the FMPLL state machine will be in the FMPLL_LOCKED state.

The CDRPLL state machine will control the CDRPLL start up and lock sequence. In order to enable the CDRPLL state machine, the MXVR_CDRPLL_CTL register should normally be programmed with the CDRCPSEL set to 0xFF and the CDRSMEN bit set to 1 and all other fields set to their reset values. Once the CDRPLL has started up, has locked to the MXI clock and is operating at the network frequency 1024 * Fs, the PFL bit in the MXVR_INT_STAT_0 register will be set to 1 and can optionally be

enabled to generate a Status Change Interrupt. The current state of the CDRPLL state machine can also be monitored by looking at the CDRPLLST field in the MXVR_STATE_0 register. At the point when the PFL bit is asserted due to CDRPLL locking at the network frequency, the CDRPLL state machine will be in the CDRPLL_FHOLD state.

The FMPLL and CDRPLL state machines can be enabled one after the other or simultaneously (for a faster overall lock time); however, it should be noted that the PFL event asserts once for frequency lock of the FMPLL and a second time for frequency lock of the CDRPLL.

Once the FMPLL and CDRPLL are locked at the network frequency (FMPLL state machine in the FMPLL_LOCKED state and the CDRPLL state machine in the CDRPLL_FHOLD state), the MXVR output clocks can be enabled and the MXVR can be enabled in Master mode to start the process of locking the MOST® network. The MXVR should never be enabled in Master mode without the FMPLL and CDRPLL having been started up and locked at frequency.

Slave Mode Initialization and Start Up of MXVR CDRPLL

Once the clock supplied on the MXI is stable and at frequency, the MXVR CDRPLL can be started up. Prior to starting up the CDRPLL, be sure that the MPS field in the MXVR_CLK_CTL register is programmed properly based on the SCLK frequency as this prescaler is used to control the PLL state machine lock times.

The CDRPLL state machine will control the CDRPLL start up and lock sequence. In order to enable the CDRPLL state machine, the MXVR_CDRPLL_CTL register should normally be programmed with the CDRCPSEL set to 0xFF and the CDRSMEN bit set to 1 and all other fields set to their reset values. Once the CDRPLL has started up, has locked to the MXI clock, and is operating at the network frequency 1024 * Fs, the PFL bit in the MXVR_INT_STAT_0 register will be set to 1 and can optionally be

enabled to generate a Status Change Interrupt. The current state of the CDRPLL state machine can also be monitored by looking at the CDRPLLST field in the MXVR_STATE_0 register. At the point when the PFL bit is asserted due to CDRPLL locking at the network frequency, the CDRPLL state machine will be in the CDRPLL_FHOLD state.

Once the CDRPLL is locked at the network frequency (the CDRPLL state machine in the CDRPLL_FHOLD state), the MXVR can be enabled in Slave mode to start the process of locking in the MOST® network. The MXVR should never be enabled in Slave mode without the CDRPLL having been started up and locked at frequency.

Enabling MXVR Output Clocks

Once the FMPLL or the CDRPLL have been started up and are frequency locked, the MXVR output clocks MMCLK, MBCLK, and MFS can programmed and enabled. The MXVR output clocks can either be generated by the FMPLL or by the CDRPLL depending on how the CLKX3SEL bit in the MXVR_CLK_CTL register is programmed. If the CLKX3SEL bit is set to 1, the MXVR output clocks are generated by the FMPLL which is locked to the frequency of the MXI input clock. If the CLKX3SEL bit is set to 0, the MXVR output clocks are generated by the CDRPLL which is either locked to the frequency of the MXI input clock (when the CDRPLL is in frequency multiply mode) or locked to the frequency of the incoming data stream (when the CDRPLL is in clock/data recovery mode).

The following steps should be followed to program the MXVR output clocks once MXVR PLL which is to generate the clocks is frequency locked:

1. MMCLKEN, MBCLKEN, and MFSEN in the MXVR_CLK_CTL register should all be set to 0 (disabling the MXVR output clocks).
2. Ensure that the MXVR is selected in the GPIO pin multiplexing for the PORTC_1 (MMCLK) and PORTC_5 (MBCLK) pins. Also, ensure that the MFSGPSEL bit is set to 0 and the MFSOE bit is set to 1 in the MXVR_PIN_CTL register.
3. Write the MMCLKMUL, MBCLKDIV, MFSDIV, MFSSEL, and MFSSYNC fields in the MXVR_CLK_CTL register to define the frequency and relationship of the MXVR output clocks.
4. Wait 1 µsec.
5. Write to the MMCLKEN, MBCLKEN, and MFSEN bits in the MXVR_CLK_CTL register to enable the individual MXVR output clocks to start toggling as desired. Set MMCLKEN to 1 to enable MMCLK. Set MBCLKEN to 1 to enable MBCLK. Set MFSEN to 1 to enable MFS.

Network Lock

The steps for enabling the MXVR and attempting to lock the MOST® network are slightly different for Master and Slave nodes. The steps for a Master node and the steps for a Slave node are described in the following sections.

Network Lock for a Master Node

Once the FMPLL and CDRPLL are at frequency, the MXVR can be enabled, the MXVR transmit can be enabled, and the PHY Transmitter can be turned on. This is accomplished by writing to the MXVR_CONFIG register and setting the MXVREN bit to 1, the MTXEN bit to 1, and the MTXONB bit

to 0. All other bits in the MXVR_CONFIG register should be left programmed as described in "Node Initialization" on page 29-111. Once the MXVR_CONFIG register has been written, the MXVR begins transmitting the network datastream on the MTX output pin and the PHY Transmitter transmits the datastream to the first slave node in the network and it begins its lock sequence.

Once the MXVR master node starts to receive incoming network activity (circling the ring network) on MRX, the CDRPLL attempts to lock onto the incoming datastream and recover the clock and data. The CDRPLL state machine completely controls this process. The FLOCK, BLOCK, and SBLOCK bits in the MXVR_STATE_0 register indicate the current network lock level of the MXVR. In addition, state changes on the lock level state bits are indicated by the FU2L, FL2U, BU2L, BL2U, SBU2L, and SBL2U interrupt event bits in the MXVR_INT_STAT_0 register. These interrupt events can generate an MXVR Status interrupt if enabled.

In the Master node, the MXVR CDRPLL charge pump current is initialized to the highest possible charge pump current value of 0xFF. This value does not need to be changed based on the number of nodes in the network as is the case in the Slave node.

Whenever the MXVR loses frame lock, the CDRPLL state machine causes the CDRPLL to relock to the clock supplied on the MXI pin and stabilize at the network frequency 1024 * Fs before attempting to relock onto the incoming datastream. This process is completely handled in hardware by the CDRPLL state machine and no software intervention is required when re-locking the network after an unlock or ring break.

Network Lock For a Slave Node

Once the CDRPLL is at frequency and the slave node has detected network activity, the MXVR can be enabled, the MXVR transmit can be enabled, and the PHY Transmitter can be turned on. This is accomplished by writing to the MXVR_CONFIG register and setting the MXVREN bit to 1, the MTXEN bit to 1, and the MTXONB bit to 0. All other bits in the MXVR_CONFIG

register should be left programmed as described in "Node Initialization" on page 29-111. Once the MXVR_CONFIG register has been written, the MXVR begins transmitting the network datastream on the MTX output pin and the PHY Transmitter transmits the datastream to the next slave node in the network and it begins its lock sequence.

Once the MXVR is enabled, the CDRPLL attempts to lock onto the incoming datastream and recover the clock and data. The CDRPLL state machine completely controls this process. The FLOCK, BLOCK, and SBLOCK bits in the MXVR_STATE_0 register indicate the current network lock level of the MXVR. In addition state changes on the lock level state bits are indicated by the FU2L, FL2U, BU2L, BL2U, SBU2L, and SBL2U interrupt event bits in the MXVR_INT_STAT_0 register. These interrupt events can generate an MXVR Status interrupt if enabled.

In the Slave node, the MXVR CDRPLL charge pump current should be adjusted for best PLL jitter performance based on the position of the slave in the network. In the Slave, whenever the PRU event occurs and the PVALID bit is asserted, the CDRPLL MCPSEL field should be programmed based on the following formula:

```
MCPSEL = POSITION << 4
```

This formula applies for networks with 16 nodes or less. Therefore, the CDRPLL MCPSEL value in the Slave nodes will range from 0x10 to 0xF0. For networks with more than 16 nodes, contact Analog Devices, Inc. for further details.

Whenever the MXVR loses frame lock, the CDRPLL state machine causes the CDRPLL to relock to the clock supplied on the MXI pin and stabilize at the network frequency 1024 * Fs before attempting to relock onto the incoming datastream. This process is completely handled in hardware by the CDRPLL state machine and no software intervention is required when re-locking the network after an unlock or ring break.

Network Initialization

Once the network is locked, the Master node typically changes the value of the MSB in the MXVR_CONFIG register. Once the MSB field is changed, the Master will distribute the new synchronous boundary over the network. The update of the RSB value in each of the slave nodes and in the master node is used to indicate that the network lock is stable and the ring is closed. In the slave nodes, the update of the RSB value will cause the SBU interrupt event to be asserted. Note that once the network is in operation, a special procedure must be followed to dynamically change the synchronous boundary without disrupting the asynchronous packet channel.

The MXVR will automatically determine the node position, node delay, maximum node position, and maximum node delay from the network. Once the lock level of the MXVR is such that these values can be determined, the fields and valid bits in the MXVR_POSITION, MXVR_DELAY, MXVR_MAX_POSITION, and MXVR_MAX_DELAY registers will be updated. In addition, the PRU, DRU, MPRU, and MDRU interrupt events in the MXVR_INT_STAT_0 register will assert when these values become valid or change.

The MXVR will also automatically receive the Allocation Table distributed by the Master node in the MXVR_ALLOC_x registers. The Master node in the network distributes its Allocation Table once every 1024 frames. Once the distribution of the Allocation Table has completed, the ATU interrupt event will assert. The assertion of the ATU interrupt event only indicates that the Allocation Table distribution is received and does not indicate whether the Allocation Table has changed since the last distribution. In the Master node, the ATU interrupt event also asserts when a Resource Allocate or Resource De-Allocate control message has caused the Allocation Table to be updated.

For the logical address to be used in the address comparison for received control messages and receive asynchronous packets, the LADDR field should be written and the LVALID bit should be set to 1 in the MXVR_LADDR register. Note that software must determine the uniqueness of the logical address.

For the group address to be used in the address comparison for received control messages, the GADDRL field should be written and the GVALID bit should be set to 1 in the MXVR_GADDR register.

For the alternate address to be used in the address comparison for received asynchronous packets, the AADDR field should be written and the AVALID bit should be set to 1 in the MXVR_AADDR register.

To enable the reception of Normal control messages the NCMRXEN bit in the MXVR_CONFIG register must be set to 1. Any Normal control message addressed to the MXVR while the NCMRXEN bit is set to 0 will not be received in the CMRB and the transmission status response "Not Supported" will be given.

To enable the reception of Remote Write control messages, the RWRRXEN bit in the MXVR_CONFIG register must be set to 1. Any Remote Write control message addressed to the MXVR while the RWRRXEN bit is set to 0 will not update the RRDB and the transmission status response "Not Supported" will be given.

To enable the reception of asynchronous packets, the APRXEN bit in the MXVR_CONFIG register must be set to 1. Any asynchronous packet addressed to the MXVR while the APRXEN bit is set to 0 will not be received in the APRB.

Synchronous Data Routing and Muting

The MXVR can either re-transmit incoming synchronous data on the same outgoing physical channel as it was received on or re-transmit incoming synchronous data on one or more different outgoing physical channels. The process of either forwarding data on the same physical channel or re-transmitting it on different physical channels is called routing. The MXVR can also mute individual synchronous data physical channels and can automatically mute physical channels when a synchronous data DMA channel has completed transmission. When a physical

channel is muted, the data 0x00 is transmitted on that physical channel. The routing and muting functions for each synchronous data physical channel are specified in the MXVR Routing registers.

The MXVR Routing registers must always be initialized prior to enabling the MXVR. All outgoing synchronous data physical channels which will not have data routed onto them should be programmed to forward the data from the corresponding incoming synchronous data physical channel. In order to forward data from incoming synchronous data physical channel m to outgoing synchronous data physical channel m, the Transmit Channel m field in the appropriate MXVR_ROUTING_x register must be programmed with the value m. For example, to forward the incoming data on physical channel 5 to the outgoing physical channel 5, the Transmit Channel 5 field in MXVR_ROUTING_1 should be written with 0x05.

In order to use the Routing registers to route data from one incoming synchronous data physical channel to one or more outgoing synchronous data physical channels, the MXVR must be a Master enabled in Active Mode (MXVREN=1, MMSM=1, and ACTIVE=1) or must be a Slave enabled in Active Mode with the Synchronous Data Delay set to 2 frames (MXVREN=1, MMSM=0, ACTIVE=1, and SDELAY=1). In addition, a DMA channel transmitting data onto a synchronous data physical channel or the muting of a synchronous data physical channel takes precedence over any programmed routing for that physical channel. To route data from incoming synchronous data physical channel m onto outgoing synchronous data physical channel n, the Transmit Channel n field in the appropriate MXVR_ROUTING_x register must be programmed with the value m. For example, to route the incoming data on physical channel 20 to the outgoing physical channel 30, the Transmit Channel 30 field in the MXVR_ROUTING_5 register should be written with 0x14.

The Routing registers also control the muting of individual synchronous data physical channels. In order to use the muting function, the MXVR must be enabled in Active Mode (MXVREN=1 and ACTIVE=1). In addition, a DMA channel transmitting data onto a synchronous data physical channel

takes precedence over muting for that physical channel. However, when the DMA channel is stopped, the synchronous data physical channel will be muted if muting is enabled for that physical channel. Enabling muting for channels which will be transmitted on keeps junk data from echoing on the bus when the transmission stops and indicates that the channel is in use even when the DMA is not actively transmitting data. To enable muting for synchronous data physical channel m, the `Mute Channel m` bit in the appropriate `MXVR_ROUTING_x` register should be set to 1. To disable muting, the `Mute Channel m` bit should be set to 0. For example, to mute physical channel 33, the `Mute Channel 33` bit in the `MXVR_ROUTING_9` should be set to 1.

Synchronous Data Transmission

The MXVR has 8 dedicated DMA channels for synchronous data transmission and reception. These 8 DMA channels can be programmed individually for transmission or reception and can be mapped to logical channels composed of any number of physical channels in the synchronous data field of the frame.

In order to set up a DMA channel for synchronous data transmission, a Logical Channel must first be defined. A Logical Channel is a set of synchronous data physical channels on which the data will be transmitted or received. A Logical Channel can include from one physical channel up to (`RSB` * 4) physical channels in size. The MXVR supports up to 8 defined Logical Channels and the Logical Channels are identified by a number from 0 to 7. Logical Channel m is defined by writing the number m into one or more `LCHANPCx` fields in the `MXVR_SYNC_LCHAN_x` which represent the synchronous data physical channels. All `LCHANPCx` fields which have the number m written to them are part of Logical Channel m.

Once the Logical Channel is defined, the DMA channel can be configured. The MXVR has 8 DMA channels dedicated for synchronous data transmission and reception. All 8 DMA channels have the same functionality and any number of them can be used simultaneously. In order to

configure DMA channel x for transmission, the configuration bits in the MXVR_DMAx_CONFIG register should be programmed. On the first write to the MXVR_DMAx_CONFIG register, the MDMAENx bit should be set to 0, the DDx bit should be set to 0 to transmit data, the LCHANx field should be programmed with the defined Logical Channel number, the BITSWAPENx and BYSWAPENx bits should be set to select any data manipulation prior to transmission, and the MFLOWx should be programmed to either Stop Mode or Autobuffer Mode.

Note that the Synchronous Packet DMA Mode encodings of the MFLOWx field and the FIXEDPMx, STARTPATx, STOPPATx and COUNTPOSx fields are only used when the DMA channel is receiving data.

Then the address of the data buffer in L1 or L2 memory to be transmitted should be programmed to the MXVR_DMAx_START_ADDR register and the number of bytes to be transmitted should be programmed to the MXVR_DMAx_COUNT register.

Once the Logical Channel has been defined, the configuration bits, the start address, and the transfer count have been written, the DMA channel can be enabled. The DMA channel is enabled by writing the MXVR_DMAx_CONFIG register again with the configuration bits programmed to the same values as before and with the MDMAENx bit now set to 1.

When the DMA channel is enabled to transmit in Stop Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. Then the DMA channel starts fetching data from L1 memory starting at the address specified in MXVR_DMAx_START_ADDR and starts transmitting the data on the physical channels defined in the associated Logical Channel. Once the DMA has fetched half of the total number of bytes specified in MXVR_DMAx_COUNT, the HDONEx status bit is set to 1. The DMA continues to fetch data from L1 and transmits the data until the total number of bytes specified in MXVR_DMAx_COUNT has been fetched. At that point, the DONEx status bit is set to 1 and the DMA channel stops and disables itself.

When the DMA channel is enabled to transmit in Autobuffer Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. Then the DMA channel starts fetching data from L1 memory starting at the address specified in MXVR_DMAx_START_ADDR and starts transmitting the data on the physical channels defined in the associated Logical Channel. Once the DMA has fetched half of the total number of bytes specified in MXVR_DMAx_COUNT, the HDONEx status bit is set to 1. The DMA continues to fetch data from L1 and transmits the data until the total number of bytes specified in MXVR_DMAx_COUNT has been fetched. At that point, the DONEx status bit is set to 1, and the DMA channel restarts itself fetching data from the address programmed in the MXVR_DMAx_START_ADDR and using the transfer count in MXVR_DMAx_COUNT. The DMA channel continues to operate in this fashion indefinitely until it is manually disabled.

A DMA channel can be disabled manually at any time by writing the MDMAEN bit to 0, however the DMA channel will not actually stop until the start of the next frame. The associated DMAACITVEx bit in the MXVR_STATE_1 register can be monitored to see when the DMA channel has actually stopped. Note that as a general rule the Logical Channel definition in the MXVR_SYNC_LCHAN_x registers should not be changed while a DMA channel is active which uses that Logical Channel.

Synchronous Data Reception

In order to set up a DMA channel for data reception, a Logical Channel must first be defined. A Logical Channel is a set of synchronous data physical channels on which the data will be transmitted or received. A Logical Channel can include from one physical channel up to (RSB * 4) physical channels in size. The MXVR supports up to 8 defined Logical Channels and the Logical Channels are identified by a number from 0 to 7. Logical Channel m is defined by writing the number m into one or more LCHANPCx fields in the MXVR_SYNC_LCHAN_x which represent the synchronous data physical channels. All LCHANPCx fields which have the number m written to them are part of Logical Channel m.

Once the Logical Channel (from which data will be received) has been defined, the DMA channel can be configured. The MXVR has 8 DMA channels dedicated for synchronous data transmission and reception. All 8 DMA channels have the same functionality and any number of them can be used simultaneously. In order to configure DMA channel x for reception, the bits in the MXVR_DMAx_CONFIG register should be programmed. On the first write to the MXVR_DMAx_CONFIG, the MDMAENx bit should be set to 0, the DDx bit should be set to 1 to receive data, the LCHANx field should be programmed with the defined Logical Channel number, the BITSWAPENx and BYSWAPENx bits should be set to select any data manipulation prior to transmission, the MFLOWx should be programmed, and the synchronous packet configuration bits should be programmed if a synchronous packet flow mode is selected.

Then the address of the data buffer in L1 or L2 memory should be programmed to the MXVR_DMAx_START_ADDR register and the MXVR_DMAx_COUNT register should be programmed based on the flow mode selected.

Once the Logical Channel has been defined, the configuration bits, the start address, and the transfer count have been written, the DMA channel can be enabled. The DMA channel is enabled by writing the MXVR_DMAx_CONFIG again with the configuration bits programmed to the same values as before and with the MDMAENx now set to 1.

When the DMA channel is enabled to receive in Stop Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. Then the DMA channel starts receiving the data on the physical channels defined in the associated Logical Channel and starts writing the data to L1 memory starting at the address specified in MXVR_DMAx_START_ADDR. Once the DMA channel has written half of the total number of bytes specified in MXVR_DMAx_COUNT, the HDONEx status bit is set to 1. The DMA continues to receive the data and write the data to L1 memory until the total number of bytes specified in MXVR_DMAx_COUNT has been written. At that point, the DONEx status bit is set to 1 and the

DMA channel stops and disables itself. See DMA channels 3, 4, and 5 in Listing 29-1 on page 29-130 for examples of a DMA channel being programmed for Stop Mode.

When the DMA channel is enabled to receive in Autobuffer Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. Then the DMA channel starts receiving the data on the physical channels defined in the associated Logical Channel and starts writing data to L1 memory starting at the address specified in MXVR_DMAx_START_ADDR. Once the DMA channel has written half of the total number of bytes specified in MXVR_DMAx_COUNT, the HDONEx status bit is set to 1. The DMA continues to receive the data and to write data to L1 until the total number of bytes specified in MXVR_DMAx_COUNT has been fetched. At that point, the DONEx status bit is set to 1, and the DMA channel restarts itself writing data to the address programmed in the MXVR_DMAx_START_ADDR and using the transfer count in MXVR_DMAx_COUNT. The DMA channel continues to operate in this fashion indefinitely until it is manually disabled.

When the DMA channel is enabled to receive in Synchronous Packet-Variable Count Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. The DMA channel then monitors incoming data in the Logical Channel looking for the occurrence of the Start Pattern as defined by the STARTPATx field and the FIXEDPMx bit. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in the associated Logical Channel and starts writing data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR. The DMA channel then finds the transfer count in the received data itself in the position determined by the COUNTPOSx field. Once the DMA channel has written the total number of bytes specified by the transfer count found in the packet, the HDONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in

the associated Logical Channel and will start writing the data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The DMA channel then finds the transfer count in the received data itself in the position determined by the COUNTPOSx field. Once the DMA channel has written the total number of bytes specified by the transfer count found in the packet, the DONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. The third packet received is written to the address specified by MXVR_DMAx_START_ADDR and the fourth packet is written to the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT, and so on. The DMA channel continues to operate in this fashion indefinitely until it is manually disabled. See DMA channel 2 in Listing 29-1 on page 29-130 for an example of a DMA channel being programmed for Synchronous Packet-Variable Count Mode.

When the DMA channel is enabled to receive in Synchronous Packet-Start/Stop Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. The DMA channel then monitors incoming data in the Logical Channel looking for the occurrence of the Start Pattern as defined by the STARTPATx field and the FIXEDPMx bit. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in the associated Logical Channel and will start writing data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR. The DMA channel then monitors incoming data in the Logical Channel looking for the occurrence of the Stop Pattern as defined by the STOPPATx field and the FIXEDPMx bit. Once the DMA channel finds the Stop Pattern, the HDONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in the associated Logical Channel and will start writing the data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The DMA channel then monitors incoming data in the Logical Channel

looking for the occurrence of the Stop Pattern as defined by the STOPPATx field and the FIXEDPMx bit. Once the DMA channel finds the Stop Pattern, the DONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. The third packet received is written to the address specified by MXVR_DMAx_START_ADDR and the fourth packet will be written to the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT, and so on. The DMA channel continues to operate in this fashion indefinitely until it is manually disabled. See DMA channel 3 in Listing 29-1 on page 29-130 for an example of a DMA channel being programmed for Synchronous Packet-Start/Stop Mode.

When the DMA channel is enabled to receive in Synchronous Packet-Fixed Count Mode, the DMA channel waits until the start of the next frame so that the data is always frame aligned. The DMA channel then monitors incoming data in the Logical Channel looking for the occurrence of the Start Pattern as defined by the STARTPATx field and the FIXEDPMx bit. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in the associated Logical Channel and starts writing data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR. The DMA channel continues to receive the data and write the data to L1 memory until the total number of bytes specified in MXVR_DMAx_COUNT has been written. At that point, the HDONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. Once the Start Pattern is found, the DMA channel then starts receiving the data on the physical channels defined in the associated Logical Channel and starts writing the data to L1 memory starting at the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The DMA channel continues to receive the data and write the data to L1 memory until the total number of bytes specified in MXVR_DMAx_COUNT has been written. At that point, the DONEx status bit is set to 1. The DMA channel then stops and will again start monitoring the incoming data in the Logical Channel looking for the occurrence of the Start Pattern. The third packet received will be written to the address

specified by MXVR_DMAx_START_ADDR and the fourth packet will be written to the address specified by MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT, and so on. The DMA channel continues to operate in this fashion indefinitely until it is manually disabled. See DMA channel 4 in Listing 29-1 on page 29-130 for an example of a DMA channel being programmed for Synchronous Packet-Fixed Count Mode.

A DMA channel can be disabled manually at any time by writing the MDMAEN bit to 0; however, the DMA channel will not actually stop until the start of the next frame. The associated DMAACITVEx bit in the MXVR_STATE_1 register can be monitored to see when the DMA channel has actually stopped. Note that as a general rule, the Logical Channel definition in the MXVR_SYNC_LCHAN_x registers should not be changed while a DMA channel which uses that Logical Channel is active.

Listing 29-1. Synchronous Packet Modes Code Example

```
// Declare and define the sync packet transmit buffers.
U8_t   tx_sync_pkt_vc[16] = {0xF0,0xF1,0xF2,0xF3,
                   0xAA,0x00,0x0C,0xAA,
                   0x01,0x02,0x03,0x04,
                   0x05,0x06,0x07,0x08};

U8_t   tx_sync_pkt_ss[16] = {0xF0,0xF1,0xF2,0xF3,
                   0x10,0x11,0x12,0x13,
                   0x14,0x15,0x16,0x17,
                   0xFF,0xFF,0xFF,0xFF};

U8_t   tx_sync_pkt_fc[20] = {0xFF,0xFF,0xFF,0xFF,
                   0x20,0x21,0x22,0x23,
                   0x24,0x25,0x26,0x27,
                   0x28,0x29,0x2A,0x2B,
                   0x2C,0x2D,0x2E,0x2F};

// Declare the sync packet receive buffers.
```

```
 U8_t   rx_sync_pkt_vc[32];
U8_t   rx_sync_pkt_ss[32];
 U8_t   rx_sync_pkt_fc[32];

 // Initialize the sync packet receive buffers.
 for (i=0; i<32; i++)
   {
   rx_sync_pkt_vc[i] = 0x00;
   rx_sync_pkt_ss[i] = 0x00;
   rx_sync_pkt_fc[i] = 0x00;
   }

// Set up the routing registers to mute the
 // physical channels which will be used for
 // Logical Channels 0, 1, and 2.
 *pMXVR_ROUTING_0 = 0x83828180;
  *pMXVR_ROUTING_1 = 0x87868584;
 *pMXVR_ROUTING_2 = 0x8B8A8988;

 // Define Logical Channels 0, 1, and 2.
 *pMXVR_SYNC_LCHAN_0 = 0x11110000;
 *pMXVR_SYNC_LCHAN_1 = 0xFFFF2222;

 // Set up the Pattern Registers
 *pMXVR_PAT_DATA_0 = 0xF0F1F2F3;
 *pMXVR_PAT_EN_0 = 0xFFFFFFFF;

 *pMXVR_PAT_DATA_1 = 0xFFFFFFFF;
 *pMXVR_PAT_EN_1 = 0xFFFFFFFF;

 // Configure and enable DMA channels 0, 1, and 2 in
 // each of the synchronous packet reception modes
 // using Logical Channels 0, 1, and 2.
```

```
*pMXVR_DMA0_CONFIG = STARTPAT_0 | COUNTPOS_1| MFLOW_PVC |
    LCHAN_0 | DD_RX;
 *pMXVR_DMA0_START_ADDR = (void*) rx_sync_pkt_vc;
*pMXVR_DMA0_COUNT = (U16_t) 16;
*pMXVR_DMA0_CONFIG |= MDMAEN;

 *pMXVR_DMA1_CONFIG = STARTPAT_0 | STOPPAT_1 | MFLOW_PSS |
    LCHAN_1 | DD_RX;
 *pMXVR_DMA1_START_ADDR = (void*) rx_sync_pkt_ss;
*pMXVR_DMA1_COUNT = (U16_t) 16;
*pMXVR_DMA1_CONFIG |= MDMAEN;

 *pMXVR_DMA2_CONFIG = STARTPAT_1 | MFLOW_PFC | LCHAN_2 | DD_RX;
*pMXVR_DMA2_START_ADDR = (void*) rx_sync_pkt_fc;
 *pMXVR_DMA2_COUNT = (U16_t) 16;
 *pMXVR_DMA2_CONFIG |= MDMAEN;

// Wait for DMA channels 0, 1, and 2 to start up in
// the synchronous packet / pattern matching mode.
while (!(*pMXVR_STATE_1 & DMAPMEN0));
 while (!(*pMXVR_STATE_1 & DMAPMEN1));
while (!(*pMXVR_STATE_1 & DMAPMEN2));

// Setup and enable DMA channels to transmit sync packets in
// Stop Mode on Logical Channels 0, 1, and 2, so they will be
// received by DMA channels 0, 1, and 2.
 *pMXVR_DMA3_CONFIG = MFLOW_STOP | LCHAN_0 | DD_TX;
 *pMXVR_DMA3_START_ADDR = (void*) tx_sync_pkt_vc;
 *pMXVR_DMA3_COUNT = (U16_t) 16;
 *pMXVR_DMA3_CONFIG |= MDMAEN;

 *pMXVR_DMA4_CONFIG = MFLOW_STOP | LCHAN_1 | DD_TX;
*pMXVR_DMA4_START_ADDR = (void*) tx_sync_pkt_ss;
 *pMXVR_DMA4_COUNT = (U16_t) 16;
```

```
 *pMXVR_DMA4_CONFIG |= MDMAEN;

 *pMXVR_DMA5_CONFIG = MFLOW_STOP | LCHAN_2 | DD_TX;
 *pMXVR_DMA5_START_ADDR = (void*) tx_sync_pkt_fc;
*pMXVR_DMA5_COUNT = (U16_t) 20;
*pMXVR_DMA5_CONFIG |= MDMAEN;

// Wait for first set of packets to be received on
// DMA channels 0, 1, and 2 and for the HDONE bits
// to be set.
 while (!(*pMXVR_INT_STAT_1 & HDONE0));
while (!(*pMXVR_INT_STAT_1 & HDONE1));
while (!(*pMXVR_INT_STAT_1 & HDONE2));
 *pMXVR_INT_STAT_1 = HDONE0 | HDONE1 | HDONE2;

// Re-transmit the same 3 sync packets again.
 *pMXVR_DMA3_CONFIG |= MDMAEN;
*pMXVR_DMA4_CONFIG |= MDMAEN;
 *pMXVR_DMA5_CONFIG |= MDMAEN;

// Wait for second set of packets to be received on
// DMA channels 0, 1, and 2 and for the DONE bits to
// be set.
while (!(*pMXVR_INT_STAT_1 & DONE0));
while (!(*pMXVR_INT_STAT_1 & DONE1));
while (!(*pMXVR_INT_STAT_1 & DONE2));
 *pMXVR_INT_STAT_1 = DONE0 | DONE1 | DONE2;
```

Asynchronous Packet Transmission

The MXVR Asynchronous Packet Transmit Buffer (APTB) is an area of memory that is allocated to hold an asynchronous packet to be transmitted. The APTB must reside in L1 or L2 memory and the starting address of the APTB is programmed in the MXVR_APTB_START_ADDR register. Enough memory should be allocated for the largest asynchronous packet to be transmitted. The largest allowed asynchronous packet data length is 1014 bytes and with 12 bytes for packet priority, addressing, and length fields, the APTB must be 1026 bytes.

Once the asynchronous packet to be transmitted is written to the APTB, the STARTAP bit in the MXVR_AP_CTL register should be set to 1 to trigger the MXVR to begin arbitration and transmission of the asynchronous packet. At that point the MXVR will start DMA'ing the asynchronous packet from the APTB into the MXVR and will begin arbitrating for the asynchronous packet channel.

While the MXVR is still arbitrating for the asynchronous packet channel, the asynchronous packet transmission can be cancelled by setting the CANCELAP bit to 1 in the MXVR_AP_CTL register. Once the STARTAP bit is set to 1, the APTB cannot be written until either the asynchronous packet is successfully sent or is successfully cancelled.

The asynchronous packet is said to be successfully sent if the MXVR wins the arbitration for the asynchronous packet channel, and transmits the packet. Once the packet is transmitted, the MXVR will set the APTS bit in the MXVR_INT_STAT_1 register and an interrupt can be conditionally generated.

The asynchronous packet is said to be successfully cancelled if the CANCELAP bit is set to 1 prior to the MXVR winning arbitration for the asynchronous packet channel. If the asynchronous packet is successfully cancelled, the MXVR will set the APTC bit in the MXVR_INT_STAT_1 register and an interrupt can conditionally be generated.

An asynchronous packet to be transmitted is DMA'ed from the APTB in L1 or L2 memory to the MXVR. The asynchronous packet contains the following fields: AP Priority, AP Destination Address, AP Length, AP Source Address, and AP Data. These asynchronous packet fields will be stored at the address offsets given in Table 29-12.

Table 29-12. Asynchronous Packet Transmit Buffer Field Offsets

APTB Address Offsets	Field Name
0x000	AP Priority
0x001	Reserved
0x002	AP Destination Address (Upper Byte)
0x003	AP Destination Address (Lower Byte)
0x004	AP Length (in quadlets)
0x005	Reserved
0x006	AP Source Address (Upper Byte)
0x007	AP Source Address (Lower Byte)
0x008 to AP Data End Offset	AP Data

The AP Priority can be any value from 0x01 to 0x0F with 0x01 being the highest priority and 0x0F being the lowest priority. The AP Priority value determines how soon after winning arbitration and transmitting an asynchronous packet will the node attempt to win arbitration again. The AP Priority value indicates the number of free frames the node will allow to pass before attempting to arbitrate again. Note that the AP Priority value self-limits the maximum possible bandwidth a node will get on the asynchronous packet channel. For example, if a node sends repeated asynchronous packets with an AP Priority value of 0x0F, the node will get 15 times less bandwidth on the asynchronous packet channel than if the AP Priority value was 0x01

In the actual arbitration process itself when more than one node is arbitrating for the asynchronous packet channel and the asynchronous packet channel is free for more than one frame, the node with the lowest POSITION will win arbitration. A node which has won arbitration is not allowed to arbitrate on the first free frame after it has transmitted. In the case when more than one node is arbitrating for the asynchronous packet channel after another node has just completed transmitting an asynchronous packet, the next downstream node which is arbitrating will win the arbitration.

The AP Destination Address should be programmed to be the logical address or alternate address of the node that will receive the asynchronous packet.

Software must calculate the AP Length field based on the length of the asynchronous packet data being transmitted. The AP Length field is a length in quadlets and the value includes 6 bytes for the AP Source Address (2 bytes) and AP CRC (4 bytes). The AP Length field can be calculated based on the length of the AP Data field (in bytes) using the following formula:

$$AP\ Length = ((Length(AP\ Data)) + 6) \div 4$$

The AP Source Address can be programmed to be any address representing the transmitting node. However, it is recommended that the logical address of the transmitting node be use.

The AP Data field contains the data to be transmitted in the asynchronous packet. The amount of data transmitted can be from 1 byte to 1014 bytes.

Asynchronous Packet Reception

The MXVR Asynchronous Packet Receive Buffer (APRB) is an area of memory that is allocated to hold received asynchronous packets. The APRB must reside in L1 Memory and the starting address of the APRB is programmed in the MXVR_APRB_START_ADDR register. Enough memory should

be allocated for two 1024-byte asynchronous packets to be stored (2048 total bytes). The asynchronous packets are of variable length (ranging from 8 bytes to 1024 bytes) so the Length of Data field must be read to determine where the end of each asynchronous packet is located.

As asynchronous packets are received by the MXVR the packets will be DMA'd into the APRB in a sequential manner (wrapping from the end back to the start). For example, APRB Entry 0 will be filled first, then APRB Entry 1, and then APRB Entry 0, etc. As each message is received, the corresponding APRBEx bit in the MXVR_AP_CTL register will be set to 1 by the MXVR indicating that receive buffer entry number x is full. Once software has read the asynchronous packet, the APRBEx bit should be cleared by writing a 1 to the corresponding bit position indicating that receive buffer entry x is now empty.

If a new asynchronous packet is arriving and the next sequential entry is full, the Asynchronous Packet Receive Buffer Overflow (APROF) bit in the MXVR_INT_STAT_0 register will be set to 1 and can conditionally generate an interrupt. The incoming packet which caused the overflow will be lost.

The two APRB entries are stored as address offsets to the APRB start address programmed in MXVR_APRB_START_ADDR register. The address offsets for the two APRB Entries are given in Table 29-13.

Table 29-13. Asynchronous Packet Receive Buffer Entry Offsets

APRB Entry Offset	APRB Entry Number
MXVR_APRB_START_ADDR + 0x000	AP Receive Buffer Entry 0
MXVR_APRB_START_ADDR + 0x400	AP Receive Buffer Entry 1

Received asynchronous packets are DMA'd to the next sequential APRB entry in L1 or L2 memory. The asynchronous packet contains the following fields: AP Destination Address, AP Length, AP Source Address, and AP Data. These asynchronous packet fields will be stored at the address offsets given in Table 29-14. Note that the end of the AP Data field is

determined by the AP Length field that was received in the packet. The AP Length field is a length in quadlets and the value includes 6 bytes for the AP Source Address (2 bytes) and AP CRC (4 bytes). The address offset of the final byte of the AP Data field is calculated as follows:

$$AP\ Data\ End\ Offset = (4 \times AP\ Length) + 3$$

Table 29-14. Asynchronous Packet Receive Buffer Entry Field Offsets

APRB Entry Address Offsets	Field Name
0x00	AP Destination Address (Upper Byte)
0x01	AP Destination Address (Lower Byte)
0x02	AP Length (in quadlets)
0x03	Reserved
0x04	AP Source Address (Upper Byte)
0x05	AP Source Address (Lower Byte)
0x06 to AP Data End Offset	AP Data

Control Message Transmission

The MXVR Control Message Transmit Buffer (CMTB) is an area of memory that is allocated to hold a control message to be transmitted. The CMTB must reside in L1 or L2 memory and the starting address of the CMTB is programmed in the MXVR_CMTB_START_ADDR register. The CMTB must be allocated 26 bytes.

Once the control message to be transmitted is written to the CMTB, the Start Control Message Transmission (STARTCM) bit in the MXVR_CM_CTL register should be set to 1 to trigger the MXVR to begin arbitration and transmission of the control message. At that point the MXVR will DMA the control message from the CMTB into the MXVR and will begin arbitrating for the control message channel.

While the MXVR is still arbitrating for the control message channel, the control message transmission can be cancelled by setting the CANCELCM bit in the MXVR_CM_CTL register. Once the STARTCM bit is set to 1, the CMTB should not be written until either the control message is successfully sent or is successfully cancelled.

The control message is said to be successfully sent if the MXVR wins arbitration for the control message channel, transmits the message, and receives a response back from the destination node or nodes. The response received back will depend on the type of control message that was transmitted. The response received back from the destination node or nodes will be DMA'd back to the CMTB. Once the response is DMA'd back to the CMTB, the MXVR will set the CMTS bit in the MXVR_INT_STAT_0 register to 1 and an interrupt can be conditionally generated. Note that regardless of the actual response value (for example, Transmission Status) received back, the MXVR will set the CMTS bit to 1.

The control message is said to be successfully cancelled if the CANCELCM bit is set to 1 prior to the MXVR winning the arbitration for the control message channel. If the control message is successfully cancelled, the MXVR will set the CMTC bit to in the MXVR_INT_STAT_0 register to 1 and an interrupt can conditionally be generated.

There are six types of control messages: Normal, Remote Read, Remote Write, Resource Allocate, Resource De-Allocate, and Remote GetSource. All six types of control message contain the following fields: CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM Transmission Status.

The CM Priority is used in the control message arbitration process. The CM Priority can range from 0x00 to 0x0F with 0x00 being the lowest priority and 0x0F being the highest priority. If more than one node is arbitrating for the control messages channel at the same time, the control message being sent with the highest CM Priority will win the arbitration. If the control messages being sent have the same CM Priority, the node which has won arbitration the least will win the arbitration. If the control

messages have the same CM Priority and the nodes sending the control messages have won arbitration an equal amount, then the node with the lowest POSITION value will win the arbitration.

The CM Destination Address should be programmed to be the logical address, physical address, or group address of the node that will receive the control message. The byte order of the CM Destination Address is such that it can be written with a word write.

The CM Source Address can be programmed to be any address representing the transmitting node. However, it is recommended that the logical address of the transmitting node be use. The byte order of the CM Source Address is such that it can be written with a word write.

The CM Message Type field determines which type of control message is being sent. Table 29-15 gives the encodings for the six types of control messages. All other values are illegal. Message types 0x01 to 0x05 are referred to as system control messages and are handled by the receiving node completely in hardware.

Table 29-15. CM Message Type Encodings

CM Message Type	Type of Message
0x00	Normal Control Message
0x01	Remote Read Control Message
0x02	Remote Write Control Message
0x03	Resource Allocate Control Message
0x04	Resource De-Allocate Control Message
0x05	Remote GetSource Control Message

The CM Transmission Status field indicates whether the destination node successfully received the control message that was transmitted. The MXVR will DMA the Transmission Status that was received back from

the destination over the bus into the CM Transmission Status field in the CMTB. Table 29-16 gives the meaning of the transmission status values received back when single cast addressing is used.

Table 29-16. Single Cast Transmission Status Encodings

CM Transmission Status	Meaning of Transmission Status
0x0000	No Response
0x1010	Transmission Successful
0x1111	Not Supported
0x2020	CRC Error
0x2121	Receive Buffer Full

Table 29-17 gives the possible meanings of the transmission status when group cast or broadcast addressing is used (since the transmission status from each of the addressed nodes is OR'd together).

Table 29-17. Group Cast/Broadcast Transmission Status Encodings

CM Transmission Status	Meaning of Transmission Status
0x0000	No Response
0x1010	Transmission Successful
	Transmission Successful and No Response
0x1111	Not Supported
	Not Supported and No Response
	Not Supported and Transmission Successful
	Not Supported and No Response and Transmission Successful
0x2020	CRC Error
	CRC Error and No Response

Table 29-17. Group Cast/Broadcast Transmission Status Encodings (Cont'd)

CM Transmission Status	Meaning of Transmission Status
0x2121	Receive Buffer Full
	Receive Buffer Full and No Response
	Receive Buffer Full and CRC Error
	Receive Buffer Full and CRC Error and No Response
0x3030	CRC Error and Transmission Successful
	CRC Error and Transmission Successful and No Response
0x3131	Transmission Successful and Receive Buffer Full
	Not Supported and CRC Error
	Not Supported and Receive Buffer Full
	Transmission Successful and Receive Buffer Full and No Response
	Not Supported and CRC Error and No Response
	Not Supported and Receive Buffer Full and No Response
	Transmission Successful and Not Supported and CRC Error
	Transmission Successful and Not Supported and Receive Buffer Full
	Transmission Successful and Not Supported and CRC Error and No Response
	Transmission Successful and Not Supported and Receive Buffer Full and No Response

Normal Control Message Transmission

The Normal control message is used to transmit data between nodes. The `CM Priority`, `CM Destination Address`, `CM Source Address`, `CM Message Type` (0x00), and `CM Data` fields should be written to the `CMTB` at the address offsets given in Table 29-18.

The CM Data field contains the data payload to be sent from the source to the destination. For Normal control messages sent using single cast addressing all 17 bytes of the CM Data field may be used for data transmission. For Normal control messages sent using group cast or broadcast addressing, only the first 16 bytes of the CM Data field should be used for data transmission. The 17th byte should be used as a unique message ID so that the destination nodes can ignore retries once they have successfully received the Normal control message. Note that software must handle the transmission of retries by retransmitting the same Normal control message with the same message ID and checking the transmission status received back.

Once a Normal control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type and CM Data fields are DMA'd from the CMTB to the MXVR. Once the MXVR wins arbitration the Normal control message is sent over the control message channel. The transmission status from the destination node or nodes is received back by the MXVR and is DMA'd back to the CMTB. The transmission status for the Normal control message will be stored in the CM Transmission Status field of the CMTB at the address offset given in Table 29-18.

Table 29-18. Normal Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x00)
0x07 - 0x17	CM Data
0x18	CM Transmission Status

If the destination node successfully receives the Normal control message, the Normal control message will be written into one of the CMRB entries and the transmission status of "Transmission Successful" will be returned. If the destination node has reception of Normal control messages disabled (NCMRXEN=0), the Normal control message will not be written to the CMRB and the transmission status of "Not Supported" will be returned. If the destination node detects a CRC error in the Normal control message, the Normal control message will not be written to the CMRB and the transmission status of "CRC Error" will be returned. If the destination node's CMRB is full, the Normal control message will not be written to the CMRB and the transmission status of "Receive Buffer Full" will be returned. If no node responds to the Normal control message, the transmission status of "No Response" will be returned.

Remote Read Control Message Transmission

The Remote Read control message is used to read data from memory or registers in another node without disturbing the node's operation. When a Remote Read control message is sent to another MXVR node, data is read from the destination node's Remote Read Buffer (RRDB). The CM Priority, CM Destination Address, CM Source Address, CM Message Type (0x01), and CM Read Address fields should be written to the CMTB at the address offsets given in Table 29-19.

For Remote Read control messages, the CM Destination Address field should be restricted to single cast addresses.

The CM Read Address field contains the address offset in the RRDB of the destination node where data should be read from. A Remote Read control message always reads 8 bytes of data at a time. Since the RRDB is 256 bytes long the CM Read Address can range from 0x00 to 0xFF. If the CM Read Address is in the range 0xF9 to 0xFF, the reads will wrap around to the start of the RRDB. For example, if the CM Read Address is 0xFE, the 8 bytes of data returned will be from address offsets 0xFE, 0xFF, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05 in the destination node's RRDB.

Once a Remote Read control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type and CM Read Address fields are DMA'ed from the CMTB to the MXVR. Once the MXVR wins arbitration the Remote Read control message is sent over the control message channel. The data read from the RRDB and the transmission status from the destination node is received back by the MXVR and is DMA'd back to the CMTB. The Remote Read data will be stored in the CM Read Data field and the transmission status for the Remote Read control message will be stored in the CM Transmission Status field of the CMTB at the address offsets given in Table 29-19.

Table 29-19. Remote Read Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x01)
0x07	Reserved (Write 0x00)
0x08	CM Read Address
0x09	Reserved (Write 0x00)
0x0A - 0x11	CM Read Data
0x12 - 0x13	Reserved
0x14	CM Transmission Status
0x16 - 0x19	Reserved

If the destination node successfully receives the Remote Read control message and returns the data from its RRDB, the transmission status of "Transmission Successful" will be returned. If there is a CRC error in the

Remote Read control message, the data from its RRDB will not be returned and the transmission status of “CRC Error” will be returned. If no node responds to the Remote Read control message, the transmission status of “No Response” will be returned.

Remote Write Control Message Transmission

The Remote Write control message is used to write data to memory or registers in another node without disturbing the node’s operation. When a Remote Write control message is sent to another MXVR node, data is written to the destination node’s Remote Read Buffer (RRDB). The CM Priority, CM Destination Address, CM Source Address, CM Message Type (0x02), and CM Write Address and CM Write Length fields should be written to the CMTB at the address offsets given in Table 29-20.

The CM Write Address field contains the address offset in the RRDB of the destination node where data should be written to. A Remote Write control message can write from 1 to 8 bytes of data at a time. Since the RRDB is 256 bytes long the CM Write Address can range from 0x00 to 0xFF. If the CM Write Address is in the range 0xF9 to 0xFF, the writes will wrap around to the start of the RRDB if the number of bytes being written causes the address to go past 0xFF. For example, if the CM Write Address is 0xFE and 6 bytes of data are to be written, then the data will be written to address offsets 0xFE, 0xFF, 0x00, 0x01, 0x02, and 0x03 in the destination node’s RRDB.

The CM Write Length field contains the number of bytes of data to be written in the RRDB of the destination node. The CM Write Length field should be in the range from 0x01 to 0x08 (indicating the number of bytes to be written). Note that if the CM Write Length field is outside the range 0x01 to 0x08, destination node will not write the data to its RRDB.

The CM Write Data field contains the data that is to be written into the RRDB of the destination node. Regardless of whether 1 byte or 8 bytes of data are to be written to the RRDB of the destination node, the specified number of bytes of data should be written starting at the address offset given for CM Write Data.

Once a Remote Write control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type, CM Write Address, CM Write Length and CM Write Data fields are DMA'd from the CMTB to the MXVR. Once the MXVR wins arbitration, the Remote Write control message is sent over the control message channel. The transmission status from the destination node is received back by the MXVR and is DMA'd back to the CMTB. The transmission status for the Remote Write control message will be stored in the CM Transmission Status field of the CMTB at the address offset given in Table 29-20.

Table 29-20. Remote Write Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x02)
0x07	Reserved (Write 0x00)
0x08	CM Write Address
0x09	CM Write Length
0x0A - 0x11	CM Write Data
0x12 - 0x13	Reserved

Table 29-20. Remote Write Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x14	CM Transmission Status
0x16 - 0x19	Reserved

If the destination node successfully receives the Remote Write control message, the `CM Write Data` will be written to its `RRDB`, the `CM Write Address` will be written to its `RRDB Write Address` field, and the `CM Write Length` will be written to its `RRDB Write Length` field, and the transmission status of “Transmission Successful” will be returned. If the destination node has the reception of Remote Write control messages disabled (`RWRRXEN`=”0”) or if the `CM Write Length` is not in the range from 0x01 to 0x08, the `RRDB` will not be written and the transmission status of “Not Supported” will be returned. If the destination node detects a CRC error in the Remote Write control message, the `RRDB` will not be written and the transmission status of “CRC Error” will be returned. If no node responds to the Remote Write control message, the transmission status of “No Response” will be returned.

Resource Allocate Control Message Transmission

The Resource Allocate control message is used to request dynamic allocation of synchronous channels from the Master node. When a Resource Allocate control message is sent to the Master to request a certain number of channels, the Master determines whether there are enough channels available and if so allocates the channels by assigning a connection label to the channels in the Allocation Table. The connection label and the channel numbers allocated are returned to the transmitting node. All nodes in the network (including the Master itself), send Resource Allocate control messages to the Master to allocate channels. The `CM Priority`,

CM Destination Address, CM Source Address, CM Message Type (0x03), and CM Allocate Number Channels fields should be written to the CMTB at the address offsets given in Table 29-21.

For Resource Allocate control messages, the CM Destination Address field should be restricted to either the logical address or the physical address of the Master node.

The CM Allocate Number Requested field contains the number of channels that the transmitting node is requesting to be allocated. The CM Allocate Number Requested should be in the range from 0x01 to 0x08. (indicating the number of channels being requested). If more than 8 channels are needed, more than one Resource Allocate control message should be sent to the Master.

Once a Resource Allocate control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM Allocate Number Requested fields are DMA'd from the CMTB to the MXVR. Once the MXVR wins arbitration the Resource Allocate control message is sent over the control message channel. The response and transmission status from the destination node is received back by the MXVR and is DMA'd back to the CMTB. The response will be stored in the CM Allocate Status, CM Allocate Number Free, and CM Allocate Channel List and the transmission status will be stored in the CM Transmission Status field of the CMTB at the address offsets given in Table 29-21.

Table 29-21. Resource Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address

Table 29-21. Resource Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x06	CM Message Type (Write 0x03)
0x07	Reserved (Write 0x00)
0x08	CM Allocate Number Requested
0x09	Reserved (Write 0x00)
0x0A	CM Allocate Status
0x0B	CM Allocate Number Free
0x0C - 0x13	CM Allocate Channel List
0x14 - 0x15	Reserved
0x16	CM Transmission Status
0x18 - 0x19	Reserved

The destination node will return the status of the allocation request in the `CM Allocate Status`. Table 29-22 gives the meaning of the `CM Allocate Status` values. If the “Allocation Successful” response is given and there was not a CRC error in the Resource Allocate control message, the requested number of channels have been allocated. If the “Destination Busy” response is given the Master node is incapable of processing the allocation request at this time and the allocation request should be re-sent. If the “Insufficient Free Channels” response is given, then there are not enough free channels to satisfy the allocation request and therefore, the allocation was not done. If the “Allocation Request Incorrect” response is given, then the `CM Allocate Request` value was out of range (0x00 or greater than 0x08) and the allocation was not done. If the “Wrong Destination” response is given, then the Resource Allocate control message was

sent to a Slave node and the allocation was not done. Note that an MXVR Master will never respond with “Destination Busy”; however, Master nodes implemented with other transceivers may do so.

Table 29-22. CM Allocate Status Encodings

CM Allocate Status	Meaning of CM Allocate Status
0x01	Allocation Successful
0x02	Destination Busy
0x03	Insufficient Free Channels
0x04	Allocation Request Incorrect
0x05	Wrong Destination

The `CM Allocate Number Free` field will contain the number of channels which are still free after the current allocation request is processed and available to be allocated. If the Resource Allocate control message is sent to a Slave node, the `CM Allocate Number Free` response will be 0x00.

The `CM Allocate Channel List` field will contain 8 bytes representing physical channel numbers. If the `CM Allocate Status` response was “Allocation Successful”, then the first byte of the `CM Allocate Channel List` will be the first of the channels that was allocated and will be the Connection Label. If n channels were requested to be allocated (`CM Allocate Requested` = n), then the first n bytes in the `CM Allocate Channel List` will be the actual channels allocated. For example, if 3 channels were requested to be allocated and the allocation was successful, then the channel numbers stored in the `CM Allocate Channel List` at address offsets 0x0C, 0x0D, and 0x0E are the channels that were allocated.

If the destination node successfully receives the Resource Allocate control message, the transmission status of “Transmission Successful” will be returned. If the destination node detects a CRC error in the Resource Allocate control message, the allocation will not take place (even if the `CM Allocate Status` response was “Allocation Successful”) and the transmission status of “CRC Error” will be returned. If no node responds to the Resource Allocate control message, the transmission status of “No Response” will be returned.

Resource De-Allocate Control Message Transmission

The Resource De-Allocate control message is used to request dynamic de-allocation of synchronous channels from the Master node. A Resource De-Allocate control message can be sent to the Master to either de-allocate all the channels that are currently allocated or to de-allocate all the channels associated with a particular Connection Label. When a Resource De-Allocate control message is sent to the Master, the Master determines whether or not the request is valid, responds with the de-allocate status and updates the Allocation Table. All nodes in the network (including the Master itself), send Resource De-Allocate control messages to the Master to de-allocate channels. The `CM Priority`, `CM Destination Address`, `CM Source Address`, `CM Message Type` (`0x04`), and `CM De-Allocate Connection Label` fields should be written to the `CMTB` at the address offsets given in Table 29-23.

For Resource De-Allocate control messages, the `CM Destination Address` field should be restricted to either the logical address or the physical address of the Master node.

The `CM De-Allocate Connection Label` field contains either the Connection Label for the channels to be de-allocated or contains 0x7F if all channels are to be de-allocated. The `CM De-Allocate Number Requested`

should be in the range from 0x00 to the uppermost synchronous channel number or can be 0x7F. The uppermost synchronous channel number can be determined by the following formula:

Uppermost Synchronous Channel Number = (4 * RSB) - 1

Once a Resource De-Allocate control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM De-Allocate Connection Label fields are DMA'd from the CMTB to the MXVR. Once the MXVR wins arbitration the Resource De-Allocate control message is sent over the control message channel. The response and transmission status from the destination node is received back by the MXVR and is DMA'd back to the CMTB. The response will be stored in the CM De-Allocate Status field and the transmission status will be stored in the CM Transmission Status field of the CMTB at the address offsets given in Table 29-23.

Table 29-23. Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x04)
0x07	Reserved (Write 0x00)
0x08	CM De-Allocate Connection Label
0x09	Reserved (Write 0x00)
0x0A	CM De-Allocate Status
0x0B -0x0D	Reserved
0x0E	CM Transmission Status
0x10 - 0x19	Reserved

The destination node will return the status of the de-allocation request in the `CM De-Allocate Status`. Table 29-24 gives the meaning of the `CM De-Allocate Status` values. If the “De-Allocation Successful” response is given and there was not a CRC error in the Resource De-Allocate control message, the channels requested to be de-allocated have been successfully de-allocated. If the “Destination Busy” response is given the Master node is incapable of processing the de-allocation request at this time and the de-allocation request should be re-sent. If the “De-Allocation Request Incorrect” response is given, then the `CM De-Allocate Connection Label` value was out of range (greater than 0x7F) and the de-allocation was not done. If the “Wrong Destination” response is given, then the Resource De-Allocate control message was sent to a Slave node and the de-allocation was not done. Note that an MXVR Master will never respond with “Destination Busy”; however, Master nodes implemented with other transceivers may do so.

Table 29-24. CM De-Allocate Status Encodings

CM De-Allocate Status	Meaning of CM De-Allocate Status
0x01	De-Allocation Successful
0x02	Destination Busy
0x04	De-Allocation Request Incorrect
0x05	Wrong Destination

If the destination node successfully receives the Resource De-Allocate control message, the transmission status of “Transmission Successful” will be returned. If the destination node detects a CRC error in the Resource De-Allocate control message, the de-allocation will not take place (even if the CM De-Allocate Status response was “De-Allocation Successful”) and the transmission status of “CRC Error” will be returned. If no node responds to the Resource De-Allocate control message, the transmission status of “No Response” will be returned.

Remote GetSource Control Message Transmission

The Remote GetSource control message is used to determine which node is transmitting data on a particular physical channel. A Remote GetSource control message can be sent using broadcast addressing and the node which is transmitting on the channel specified in the CM GetSource Channel field will respond with its physical address, logical address and group address. In addition, by setting the CM GetSource Channel field to 0xFF and sending a Remote GetSource control message to a node using single cast addressing, the destination node will respond with its physical address, logical address, and group address. The CM Priority, CM Destination Address, CM Source Address, CM Message Type (0x04), and CM GetSource Channel fields should be written to the CMTB at the address offsets given in Table 29-25.

For Remote GetSource control messages, the CM Destination Address field should normally be sent using broadcast addressing; however, single cast addressing may be used to request a particular node to return its physical, logical and group addresses by sending 0xFF in the CM GetSource Channel field.

The CM GetSource Channel field contains a physical channel number. The CM GetSource Channel should be in the range from 0x00 to the uppermost synchronous channel number or can be 0xFF. The uppermost synchronous channel number can be determined by the following formula:

Uppermost Synchronous Channel Number = (4 * RSB) - 1

Once a Remote GetSource control message is written to the CMTB and the STARTCM bit is set to 1, the CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM GetSource Channel fields are DMA'd from the CMTB to the MXVR. Once the MXVR wins arbitration the Remote GetSource control message is sent over the control message channel. The response and transmission status from the destination node is received back by the MXVR and is DMA'd back to the CMTB. The response will be stored in the CM GetSource Physical Address (Low), CM

GetSource Group Address (Low), CM GetSource Logical Address (Low), and CM GetSource Logical Address (High) fields and the transmission status will be stored in the CM Transmission Status field of the CMTB at the address offsets given in Table 29-25.

Table 29-25. Remote GetSource Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x05)
0x07	Reserved (Write 0x00)
0x08	CM GetSource Channel
0x09	Reserved (Write 0x00)
0x0A - 0x0C	Reserved
0x0D	CM GetSource Physical Address (Low)
0x0E	Reserved
0x0F	CM GetSource Group Address (Low)
0x10	CM GetSource Logical Address (High)
0x11	CM GetSource Logical Address (Low)
0x12 - 0x13	Reserved
0x14	CM Transmission Status
0x16 - 0x19	Reserved

If the destination node is an MXVR, the destination node will respond to the Remote GetSource control message if the destination node is routing data onto or is muting the channel specified in the `CM GetSource Channel` field or if the `CM GetSource Channel` field is 0xFF. Other types of transceivers respond when the specified channel is routed.

If the destination node responds to the Remote GetSource control message, the node will return the low byte of its physical address (`POSITION`) in the `CM GetSource Physical Address (Low)` field.

If the destination node responds to the Remote GetSource control message, the node will return the low byte of its group address (`GADDRL`) in the `CM GetSource Group Address (Low)` field.

If the destination node responds to the Remote GetSource control message, the node will return its logical address (`LADDR`) in the `CM GetSource Logical Address (High)` and `CM GetSource Logical Address (Low)` fields.

If the Remote GetSource control message was sent and a response was received back and there was not a CRC error, the transmission status of "Transmission Successful" will be returned. If the Remote GetSource control message was sent and a response was received back but there was a CRC error, the transmission status of "CRC Error" will be returned. If the Remote GetSource control message was sent and no node responded, the transmission status of "No Response" will be returned. Note that since broadcast addressing is normally used when sending a Remote GetSource control message, it is possible that multiple nodes may respond and therefore, the transmitting node will receive back the response sent from the closest upstream node that responded.

Control Message Reception

The following sections describe control message reception operations.

Normal Control Message Reception

The MXVR Control Message Receive Buffer (CMRB) is an area of memory that is allocated to hold received control messages. The CMRB must reside in L1 or L2 memory and the starting address of the CMRB is programmed in the MXVR_CMRB_START_ADDR register. Enough memory should be allocated for sixteen 24-byte messages to be stored (384 total bytes).

As Normal control messages are received by the MXVR the Normal control messages will be DMA'd into the CMRB in a sequential manner (wrapping from the end back to the start). For example, CMRBE0 will be filled first, then CMRBE1, …, then CMRB15, then CMRE0, etc. As each message is received, the corresponding CMRBEx bit in the MXVR_CM_CTL register will be set to 1 by the MXVR indicating that receive buffer entry number x is full. Once software has read the Normal control message, the CMRBEx bit should be cleared by writing a 1 to the corresponding bit position indicating that receive buffer entry x is now empty.

If a new Normal control message is arriving and the next sequential entry is full, the Control Message Receive Buffer Overflow (CMRBOF) bit in the MXVR_INT_STAT_0 register will be set to 1 and can conditionally generate an interrupt. The incoming message which caused the overflow will be lost and the Transmission Status will be returned to the transmitter indicating that the receive buffer was full. Note that an overflow will occur if the next sequential entry is full regardless of whether other entries in the CMRB are empty.

The 16 CMRB Entries are stored as address offsets to the CMRB start address programmed in the MXVR_CMRB_START_ADDR register. The address offsets for the 16 CMRB Entries are given in Table 29-26.

Table 29-26. Control Message Receive Buffer Entry Offsets

CMRB Entry Offset	CMRB Entry Number
MXVR_CMRB_START_ADDR + 0x000	CM Receive Buffer Entry 0
MXVR_CMRB_START_ADDR + 0x016	CM Receive Buffer Entry 1

Table 29-26. Control Message Receive Buffer Entry Offsets

CMRB Entry Offset	CMRB Entry Number
MXVR_CMRB_START_ADDR + 0x02C	CM Receive Buffer Entry 2
MXVR_CMRB_START_ADDR + 0x16 * x	CM Receive Buffer Entry x
MXVR_CMRB_START_ADDR + 0x14A	CM Receive Buffer Entry 15

Received Normal control messages are DMA'd to the next sequential `CMRB` Entry in L1 or L2 memory. The Normal control message contains the following fields: `CM Destination Address`, `CM Source Address`, `CM Message Type`, and `CM Data`. The byte order of the `CM Destination Address` and `CM Source Address` will be swapped from the order that they were received, so that the addresses can be read properly with a word access. These Normal control message fields will be stored at the address offsets given in Table 29-27.

Table 29-27. Control Message Receive Buffer Entry Field Offsets

CMRB Entry Address Offsets	Field Name
0x00	CM Destination Address
0x02	CM Source Address
0x04	CM Message Type
0x05 - 0x15	CM Data

Remote Read and Remote Write Reception

The MXVR Remote Read Buffer (`RRDB`) is a buffer in L1 or L2 memory that is allocated to allow other nodes to remotely read from and write to the ADSP-BF54x processor over the network. When a Remote Read control message is received by the MXVR, the 8 bytes of data requested will be DMA'd from the `RRDB` into the MXVR so that the data can be sent out in response to the Remote Read control message. When a Remote Write control message is received by the MXVR, the up to 8 bytes of write data

will be DMA'd to the addresses specified in the Remote Write control message. In addition, the write address and write data length will also be written into fields in the RRDB.

The RRDB must reside in L1 or L2 memory and the starting address of the RRDB is programmed in the MXVR_RRDB_START_ADDR register. The RRDB must be allocated 258 bytes in L1 or L2 memory (256 bytes for data, one byte for the RRDB Write Address field and one byte for the RRDB Write Length field.

It is the responsibility of the software to ensure that a Remote Read control message is not in progress when updating the RRDB. When a Remote Read control message is being received, the RRDIP state bit will be asserted. Software should not write the RRDB while the RRDIP bit is asserted. The RRDIP bit asserts microseconds before the actual data read occurs and remains asserted for microseconds after the data read occurs.

When a Remote Write control message is being received, the Remote Write In Progress (RWRIP) bit in the MXVR_STATE_0 register will be asserted. The RWRIP bit will assert microseconds before the actual data write occurs. When the received CM Write Data, CM Write Address, and CM Write Length have been DMA'ed to the RRDB, the Remote Write Complete (RWRC) status bit will assert and an interrupt can be conditionally generated.

The start address of the RRDB is programmed in MXVR_RRDB_START_ADDR register. The received CM Write Data will be written into the RRDB Data field at the offset specified by the received CM Write Address. The received CM Write Address will be written into the RRDB Write Address field and the received CM Write Length will be written into the RRDB Write Length field so that when the Remote Write completes, software can easily determine which bytes have been remotely written. Table 29-28 gives the offsets of the RRDB Data, the RRDB Write Address, and the RRDB Write Length fields in the RRDB.

Table 29-28. Remote Read Buffer Field Offsets

RRDB Address Offsets	Field Name
0x000 - 0x0FF	RRDB Data
0x100	RRDB Write Address
0x101	RRDB Write Length

Resource Allocate Reception

The reception of Resource Allocate control messages by the MXVR is handled completely in hardware. No software intervention is required other than to observe changes to the Allocation Table once the Resource Allocate control message is processed by the MXVR.

If a Resource Allocate control message is received by the MXVR when in Master mode, the ALIP bit in the MXVR_STATE_0 register will change to 1 to indicate a Resource Allocate control message is being processed. While the ALIP bit is a 1, the Allocation Table should not be read since the Allocation Table may be only partially updated. The MXVR will first determine whether the allocation request is correct, which channels are currently free in the Allocation Table, and whether there are enough channels available to satisfy the request. The MXVR will respond with the appropriate CM Allocate Status, CM Allocate Number Free, and CM Allocate Channel List. If no CRC error occurs during the Resource Allocate control message, the MXVR will update its Allocation Table to reflect the allocation request. Once the Allocation Table is updated in the Master, the ATU bit in the MXVR_INT_STAT_0 register will change to 1. Note that the Master only distributes its Allocation Table to the Slave nodes once every 1024 frames.

If a Resource Allocate control message is received by the MXVR when in Slave mode, the MXVR will respond with the CM Allocate Status of "Wrong Destination".

Resource De-Allocate Reception

The reception of Resource De-Allocate control messages by the MXVR is handled completely in hardware. No software intervention is required other than to observe changes to the Allocation Table once the Resource De-Allocate control message is processed by the MXVR.

If a Resource De-Allocate control message is received by the MXVR when in Master mode, the DALIP bit in the MXVR_STATE_0 register will change to 1 to indicate a Resource De-Allocate control message is being processed. While the DALIP bit is a 1, the Allocation Table should not be read since the Allocation Table may be only partially updated. The MXVR will first determine whether the de-allocation request is correct, and which channels are currently allocated to the connection label in the request. The MXVR will respond with the appropriate CM De-Allocate Status. If no CRC error occurs during the Resource De-Allocate control message, the MXVR will update its Allocation Table to reflect the de-allocation request. Once the Allocation Table is updated in the Master, the ATU bit in the MXVR_INT_STAT_0 register will change to 1. Note that the Master only distributes its Allocation Table to the Slave nodes once every 1024 frames.

If a Resource De-Allocate control message is received by the MXVR when in Slave mode, the MXVR will respond with the CM De-Allocate Status of “Wrong Destination”.

Remote GetSource Reception

The reception of Remote GetSource control messages by the MXVR is handled completely in hardware. No software intervention is required.

If a Remote GetSource control message is received by the MXVR, the RGSIP bit in the MXVR_STATE_0 register will change to 1 to indicate a Remote GetSource control message is being processed. The MXVR will first determine whether it should respond to the Remote GetSource control message. The MXVR will respond if the MXVR is muting or routing data onto the channel specified in the CM GetSource Channel field or if

the CM GetSource Channel field is 0xFF. The MXVR is muting channel n when the Channel Mute n bit in the appropriate MXVR_ROUTING_x register is set to 1. The MXVR is routing data onto channel n when the Transmit Channel n field in the appropriate MXVR_ROUTING_x register is set to any value other than n. If the CM GetSource Channel field is 0xFF, the MXVR will always respond regardless of what channels are being muted or routed. The MXVR will not respond if the CM GetSource Channel field has a value between 4 * RSB and 0xFE.

When the MXVR responds to a Remote GetSource control message, the MXVR returns the low byte of its Physical Address, the low byte of its Group Address, and the high and low bytes of its Logical Address. Note that values in the POSITION, GADDRL, and LADDR fields are returned in the response regardless of whether the corresponding valid bits are set to 1. Note that the Remote GetSource control message is normally sent as a broadcast message, so it is possible that more than one node could respond with one response overwriting another.

MXVR Low Power Operation

The ADSP-BF54x processor processor provides a number of mechanisms for dynamically controlling performance and power dissipation. The main mechanisms are controlling the voltage level of the processor through the on-chip voltage regulator, controlling the core clock and system clock frequencies, and controlling the operating mode of the core and the system PLL. For more information see Chapter 18, “Dynamic Power Management”. Within the ADSP-BF54x processor dynamic power management framework, the MXVR has six general power/functionality states as shown in Table 29-29.

Table 29-29. ADSP-BF54x processor Power/MXVR Functionality States

ADSP-BF54x processor Power State	MXVR State	MXVR Data Rx/Tx	Core Clock	System Clock	Wake-up Source	MOST Network
Full-on Mode	Any	Yes[1]	From PLL	From PLL		Active
Active Mode	Any	Yes[1]	From CLKIN	From CLKIN		Active
Sleep Mode	Any	Yes	Disabled	From PLL	Any Interrupt	Active
Deep Sleep Mode	All Bypass - MXVR Disabled	No	Disabled	Disabled	Reset or RTC	Active
Hibernate State	Powered Down	No	Powered Down	Powered Down	Reset, RTC, $\overline{\text{MRXON}}$, CANRX	Not Active
Power Gated Off to ADSP-BF54x processor	Powered Down	No	Powered Down	Powered Down	Handled on Board-Level	Not Active

1 Core Clock frequency and System Clock frequency must be operated at a high enough frequency to support MXVR and other system DMA bandwidth to L1 or L2 memory.

These power/functionality states are listed in order from least power savings (Full On Mode) to greatest power savings (Power Gated Off). The functionality of the MXVR in each of these states is described in the following sections.

Full On Mode

When the ADSP-BF54x processor is operated in the Full On mode, the MXVR is fully functional and can be operated in any of its modes. While in the Full On mode, the system PLL generates the core clock and system clock. For power savings the core clock frequency can be reduced based on the minimum core processing performance required by the application and the system clock frequency can be reduced based on the minimum internal and external bus bandwidth required by the application. Once the

minimum core clock frequency and system clock frequency required for the application is known, the voltage level of the on-chip voltage regulator may be lowered to further reduce power consumption. The tables giving the minimum operating voltage level for a given core clock/system clock frequency combination can be found in the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

The MXVR utilizes its DMA channels to transfer data to and from L1 or L2 memory in order to transmit and receive synchronous data, asynchronous packets, and control messages. This means that the core clock and system clock frequency must be operated at a high enough frequency to support the bandwidth and latency requirements of the MXVR DMA channels in conjunction with all other L1 or L2 memory bandwidth used in the system (for example, memory DMA operations to/from L1 or L2, other peripheral DMA to/from L1 or L2, and core accesses to/from L1 or L2). Therefore, performance analysis must be done on a given application when choosing the minimum core clock and system clock frequencies. During this analysis, the MXVR's FIFO Error interrupt event (FERR) should be monitored. If the FERR interrupt ever asserts, the MXVR DMAs are being starved and data corruption may have occurred due to the lack of DMA bandwidth. If this occurs, the core clock and/or system clock frequency should be increased or other traffic to L1 or L2 memory should be reduced.

If the MXVR is going to be operated in All Bypass-MXVR Disabled mode for long periods of time while in Full On mode, the MXVR CDRPLL, the FMPLL, and the MXVR Crystal Oscillator or MXI clock input can be disabled to reduce power consumption. To disable the CDRPLL, the CDRSMEN bit should be set to 0 in the MXVR_CDRPLL_CTL register. To disable the FMPLL, the FMSMEN bit should be set to 0 in the MXVR_FMPLL_CTL register. If a crystal is connected between MXI and MXO, to disable the MXVR Crystal Oscillator the MXTALFEN and MXTALCEN bits should be set to 0 in the MXVR_CLK_CTL register to 0. If an external oscillator is used to supply the

MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_CLK_CTL register to gate off the MXI clock in the pad.

Active Mode

When the ADSP-BF54x processor is operated in the Active Mode, the MXVR is fully functional and can be operated in any of its modes. While in the Active Mode the system PLL is bypassed and the core clock and system clock run at the frequency of CLKIN. The core clock frequency must be operated at a high enough frequency to support the core processing performance required by the application and the system clock frequency must be operated at a high enough frequency to support the internal and external bus bandwidth required by the application. Based on the core clock frequency and system clock frequency, the voltage level of the on-chip voltage regulator may be lowered to further reduce power consumption. The tables giving the minimum operating voltage level for a given core clock/system clock frequency combination can be found in the *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

The MXVR utilizes its DMA channels to transfer data to and from L1 or L2 memory in order to transmit and receive synchronous data, asynchronous packets, and control messages. This means that the core clock and system clock frequency must be operated at a high enough frequency to support the bandwidth and latency requirements of the MXVR DMA channels in conjunction with all other L1 or L2 memory bandwidth used in the system (for example, memory DMA operations to/from L1 or L2, other peripheral DMA to/from L1 or L2, and core accesses to/from L1 or L2). Therefore, performance analysis must be done on a given application when choosing the minimum core clock and system clock frequencies. During this analysis, the MXVR's FIFO Error interrupt event (FERR) should be monitored. If the FERR interrupt ever asserts, the MXVR DMAs are being starved and data corruption may have occurred due to the lack

of DMA bandwidth. If this occurs, the core clock and/or the system clock frequency should be increased or other traffic to L1 or L2 memory should be reduced.

If the MXVR is going to be operated in All Bypass-MXVR Disabled mode for long periods of time while in Active Mode, the MXVR CDRPLL, the FMPLL, and the MXVR Crystal Oscillator or MXI clock input can be disabled to reduce power consumption. To disable the CDRPLL, the CDRSMEN bit should be set to 0 in the MXVR_CDRPLL_CTL register. To disable the FMPLL, the FMSMEN bit should be set to 0 in the MXVR_FMPLL_CTL register. If a crystal is connected between MXI and MXO, to disable the MXVR Crystal Oscillator the MXTALFEN and MXTALCEN bits should be set to 0 in the MXVR_CLK_CTL register to 0. If an external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_CLK_CTL register to gate off the MXI clock in the pad.

Sleep Mode

When the ADSP-BF54x processor is operated in Sleep Mode and the MXVR may be operated in any of its modes. The MXVR can transmit and receive synchronous data, asynchronous packets, and control messages as long as their associated memory buffers are located in L2 memory (accesses to L1 memory while in Sleep Mode is not allowed). Since the MOST® bus protocol is handled in hardware, the MXVR may be operated as either a master or a slave while in Sleep Mode. As the MOST® network master, the MXVR can continue to handle allocation and de-allocation system control messages and other background network functions.

Once the ADSP-BF54x processor has entered Sleep Mode, it can be woken up back into either the Full On mode or Active Mode by any system interrupt. The wake-up interrupt should be enabled within the

peripheral and within the IC_IWRx registers. Within the MXVR the interrupt sources that typically would be used to wake-up from Sleep Mode are:

- Reception of a Wake-up Preamble on the MOST® network (WUP)
- Detection of edges the $\overline{\text{MRXON}}$ input which is typically connected to the MOST® FOR Status Output (MH2L, ML2H)
- Detection of network activity changes on the MRX input (NI2A, NA2I)
- System Clock Counter, Frame Counter, or Block Counter time-outs (SCZ, FCZx, BCZ)
- Detection of network lock changes (SBU2L, SBL2U, BU2L, BL2U, FU2L, FL2U)
- Detection of network status changes (PRU, MPRU, DRU, MDRU, SBU, ATU)
- Reception of synchronous data, a control message or an asynchronous packet (HDONEx, DONEx, CMR, RWRC, APR)

Some examples of other interrupt sources that may typically be used to wake-up from Sleep Mode are:

- Real Time Clock events
- Timer time-out
- PFx pin edge or level
- Peripheral data reception or transmission

If the MXVR is going to be operated in All Bypass-MXVR Disabled mode while in Sleep Mode, the CDRPLL, the FMPLL, and the MXVR Crystal Oscillator or MXI clock input can be disabled to reduce power consumption. To disable the CDRPLL the CDRSMEN bit should be set to 0 in the

MXVR_CDRPLL_CTL register. To disable the FMPLL, the FMSMEN bit should be set to 0 in the MXVR_FMPLL_CTL register. If a crystal is connected between MXI and MXO, to disable the MXVR Crystal Oscillator the MXTALFEN and MXTALCEN bits should be set to 0 in the MXVR_CLK_CTL register. If an external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_CLK_CTL register to gate off the MXI clock in the pad.

Deep Sleep Mode

When the ADSP-BF54x processor is operated in Deep Sleep Mode, the core clock and the system clock are disabled, therefore, the MXVR may only be operated in All Bypass-MXVR Disabled mode. The MXVR is by default in the All Bypass-MXVR Disabled mode after reset, or the All Bypass-MXVR Disabled mode may be entered by writing the MXVREN bit to 0. Within the All Bypass-MXVR Disabled mode, the MRX input is directly connected to the MTX output, so the MOST® network can still be active while an ADSP-BF54x processor slave node is in Deep Sleep Mode.

Deep Sleep Mode can be exited only by a Real Time Clock interrupt or hardware reset. A Real Time Clock interrupt causes the ADSP-BF54x processor to transition to the Active Mode and the core will continue executing the code following the idle instruction. A hardware reset will cause the ADSP-BF54x processor to exit Deep Sleep Mode and begin the hardware reset booting sequence.

If a crystal is connected between MXI and MXO, the MXVR Crystal Oscillator may also be disabled to eliminate the power consumption of the crystal oscillator during Deep Sleep Mode. This is accomplished by setting the MXTALFEN and MXTALCEN bits to 0 in the MXVR_CLK_CTL register before executing the idle instruction that causes the ADSP-BF54x processor to enter Deep Sleep Mode. The reset sequence when exiting from Deep Sleep Mode will cause the MXTALFEN and MXTALCEN bits to be reset to 1, so the MXVR Crystal Oscillator will start up during the reset sequence. If an

external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_CLK_CTL register to gate off the MXI clock in the pad.

Hibernate State

When the ADSP-BF54x processor is operated in Hibernate State, the on-chip voltage regulator is turned off and the internal power supplies (VDDINT, VDDMP) transition to 0V. The only power that is used in this mode is the leakage current on the external power supplies (VDDEXT, VDDDDR, VDDUSB, VDDMC, VDDMX) and the current used by the Real Time Clock. In Hibernate State, the MXVR is completely powered off, so there is no longer a connection between the MRX input and the MTX output. Therefore, the MOST® network cannot be active while the ADSP-BF54x processor is in Hibernate State.

There are four wake-up sources that can wake the ADSP-BF54x processor from Hibernate State:

- Real Time Clock Interrupt
- Asserting Hardware Reset
- Asserting the $\overline{\text{MRXON}}$ input low (typically connected to the MOST® FOR Status output)
- Asserting the CANRX input low

Hardware reset always wakes up the ADSP-BF54x processor from hibernate state. The other hibernate wake-up sources can be individually enabled by setting bits in the VR_CTL register before executing the idle instruction that causes the ADSP-BF54x processor to enter hibernate state. In the VR_CTL register, the WAKE bit should be set to 1 to allow wake-up on Real Time Clock interrupts, the MXVRWE bit should be set to 1 to allow wake-up on the assertion of $\overline{\text{MRXON}}$. When any one of the enabled

wake-up source events occurs, the on-chip voltage regulator will turn on and the ADSP-BF54x processor will begin the hardware reset booting sequence.

If a crystal is connected between MXI and MXO, the MXVR Crystal Oscillator may also be disabled to eliminate the power consumption of the crystal oscillator during Hibernate State. This is accomplished by setting the MXTALFEN and MXTALCEN bits in the MXVR_CLK_CTL register to 0 before executing the idle instruction that causes the ADSP-BF54x processor to enter Hibernate State. The reset sequence when exiting from Hibernate State will cause the MXTALFEN and MXTALCEN bits to be reset to 1, so the MXVR Crystal Oscillator will start up during the reset sequence. If an external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_CLK_CTL register to gate off the MXI clock in the pad.

Power Gating the ADSP-BF54x processor

To achieve the lowest possible power consumption for a MOST® node, the external power supplies (VDDEXT, VDDDDR, VDDUSB, VDDMC, VDDMX, VDDRTC) to the ADSP-BF54x processor should be gated off and pulled to 0V. This effectively reduces the ADSP-BF54x processor power consumption to zero. Typically the MOST® FOR status output would be used to gate the ADSP-BF54x processor power supplies on and off based on the reception of modulated light.

30 KEYPAD INTERFACE

This chapter describes the 16-pin programmable keypad interface and includes the following sections:

- "Interface Overview" on page 30-1
- "Description of Operation" on page 30-2
- "Functional Description" on page 30-7
- "Programming Model" on page 30-9
- "Keypad Registers" on page 30-10
- "Programming Examples" on page 30-20

Interface Overview

The 16-pin programmable keypad interface features:

- Programmable input keypad matrix size
- Programmable debounce filter width
- Press-release-press mode support
- Interrupt on any key pressed capability
- Multiple key pressed detection and limited multiple key resolution capability

The keypad is a 16-pin interface module that is used to detect the key pressed in an 8x8 (maximum) keypad matrix. The size of the input keypad matrix is software programmable. The interface is capable of filtering the bounce on the input pins with a programmable width of the filtered bounce. The keypad module supports two modes of operation, press-release-press mode and Press-Hold mode. The press-release-press mode identifies a press-release-press sequence of a key as two consecutive presses of the same key. The Press-Hold mode checks the input key's state in periodic intervals to determine the number of times the same key is meant to be pressed.

The keypad interface module can be programmed to generate an interrupt request when it identifies that any key is pressed. Software can be programmed to detect simultaneous multiple key presses with limited multiple key resolution capability.

Description of Operation

The following sections describe the operation of the keypad interface.

Keypad Operation

A keypad interface consists of a matrix with two sets of wires, one set that runs horizontally (rows), and another that runs vertically (columns) with a pushbutton switch at each intersection. The row and column wires do not touch, but run over each other. When the pushbutton is pressed, a contact is established at the intersection of a given row and column serving as a switch. The number of switches for a given matrix depends on the number of rows and columns. For example, a 4x4 matrix can support up to 16 switches. A block diagram of the keypad interface is shown in Figure 30-1.

As shown in the figure, the column wires are connected to the column outputs of the keypad interface while the row wires are connected to the row inputs. Each row wire of the keypad has a pull-up resistor that pulls

the row wires high. When no key is pressed, there is no contact between any of the column drivers to the row inputs. As a result, all row inputs are read as 1. When a pushbutton is pressed, a contact is established between each corresponding row and column wire. Row inputs will sense the value driven by the column drivers. To determine which key is pressed, the column drivers drive zero. On a key press, the zero will be visible on the row inputs. The interface being aware of which column was driven with what value along with reading the row inputs, it could determine which key is pressed. The rest of the pages define and explain the infrastructure to determine the keys pressed.

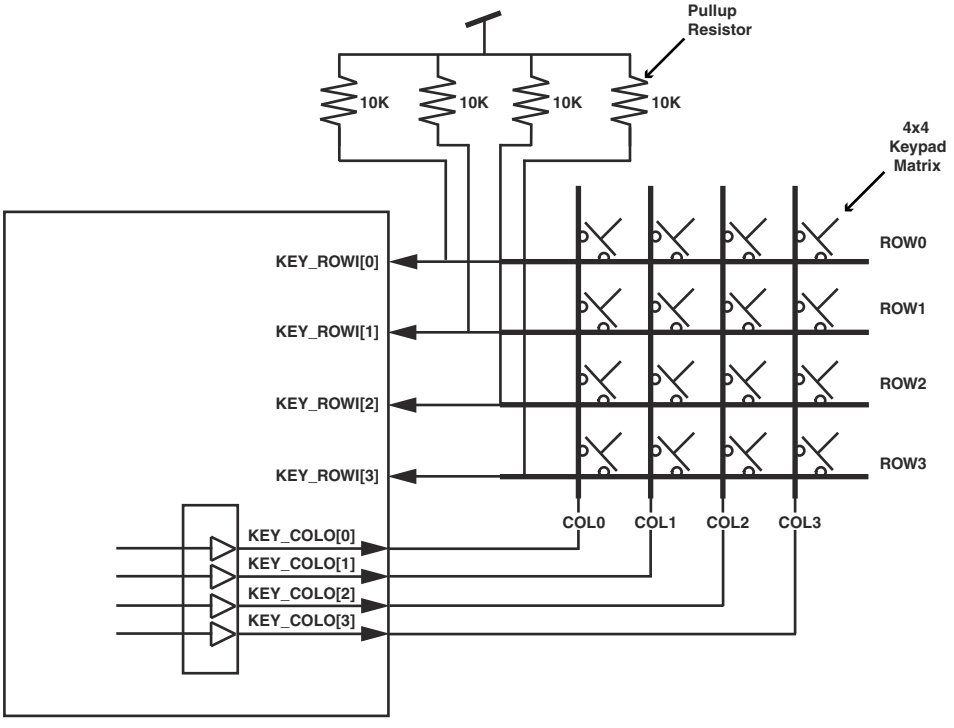


Figure 30-1. Blackfin Processor Keypad Interface

Keypad Enable/Disable

The keypad module is enabled by setting the KPAD_EN field of the KPAD_CTL register. Once enabled, the keypad module state machine drives all column outputs with a value of 0 and scans the row inputs for a key press.

The keypad module is disabled by clearing the KPAD_EN field. Clearing this field clears all input enables, counters, and drivers, and resets the state of keypad module. When the KPAD_EN field is cleared, the keypad module loses the capability to generate an interrupt request to the core.

Input Keypad Matrix Programmability

The keypad module can support a maximum of an 8x8 matrix, for a total of 64 buttons. The input keypad matrix is programmable through the KPAD_ROWEN and KPAD_COLEN fields of the KPAD_CTL register. The KPAD_ROWEN bit field is used to program the number of active rows, while KPAD_COLEN programs the number of active columns. The value of the (KPAD_ROWEN + 1) determines the number of rows enabled in the input keypad matrix. Similarly, the value of the (KPAD_COLEN + 1) determines the number of active columns.

Table 30-1. Input Keypad Matrix Programmability

KPAD_ROWEN Value	KPAD_COLEN Value	Input Keypad Matrix
b#011	b#001	4 Rows and 2 Columns
b#010	b#001	3 Rows and 2 Columns
b#111	b#111	8 Rows and 8 Columns

Waking Up on Keypad Press

When the processor is in hibernation, it can be waken up by the activity on the keypad row pins. To do that, before the processor is put into hibernation, the keypad wakeup enable (KPADWE) control bit of VR_CTL register must be set. To use the row pin KEY_ROWx to wake up the processor, its

corresponding bit in PORTx_FER register must be set to 0 (GPIO mode) and its corresponding bit in PORTx must be set to 1. Then, when an active low state is detected on the KEY_ROWx pin, a wakeup event is generated and wakes up the processor from hibernation. For more information about hibernation and GPIO, see Chapter 18, "Dynamic Power Management" and Chapter 9, "General-Purpose Ports".

Sensitivity of Keypad Interface

The sensitivity of the keypad interface to key presses is programmable through KPAD_PRESCALE and KPAD_MSEL registers. Together with the DBON_SCALE and COLDRV_SCALE fields in the KPAD_MSEL register, the value in the prescale register is used to calculate the debounce period (Tdb) and column drive period (Tcw).

Once a key press shorts the column and row wires, the debounce counter in the keypad module is triggered. The row inputs are sampled after the programmable (DBON_SCALE) debounce time of Tdb. If any of the sampled row inputs is zero, this kicks off evaluate state. The Keypad interface logic three-states all the column outputs (except one) for a pre-programmed column drive width of Tcw. With external pull-up resistors pulling up all row inputs, the Keypad interface logic pulls down one column at a time for Tcw and samples the row inputs to determine which key is pressed.

Limited Multiple Key Resolution

The keypad interface can be programmed to generate an interrupt for multiple key press detection by writing b#10 to the KPAD_IRQMODE bit field of the KPAD_CTL register (single key presses will also generate an interrupt in this mode). The KPAD_ROWCOL register records the keys pressed and can be read in the interrupt service routine for data on keys pressed. It must be noted that only certain key press combinations can be exactly resolved by reading the KPAD_ROWCOL register as follows:

- Keys pressed in a single row and a single column
- Keys pressed in a single row and a multiple columns
- Keys pressed in multiple rows and a single column

In case of keys pressed in multiple rows and multiple columns, it is not possible to predict the exact keys pressed with the existing hardware. The KPAD_MROWCOL bit field of the KPAD_STAT can be used to distinguish this scenario from the others. This bit field is set by the keypad interface when it detects key presses on multiple rows and multiple columns, allowing the user to define an action for this condition.

Keypad Interrupt Modes

The keypad interface module can be programmed to interrupt the core when it detects a key press based on the KPAD_IRQMODE bit field in the KPAD_CTL register. The KPAD_IRQMODE provides programmability to suppress interrupt generation on multiple key presses (single key presses will still generate an interrupt in this mode). Alternately, the keypad module can be programmed to interrupt the core on any key press (single or multiple key presses) on any row or column.

Implementing Press-Hold Feature

In some applications, it might be desirable to detect prolonged key presses and interpret them as multiple key presses. This feature is referred to as press-hold in this manual. The keypad module provides the KPAD_PRESSED bit field of the KPAD_STAT register to implement this feature.

After a key press is detected and the module has completed scanning for keys pressed, the keypad module interface asserts KPAD_PRESSED until the pressed key is released. If the interrupt generation is enabled (by setting the KPAD_IRQMODE bit field in the KPAD_CTL register to either b#01 or b#10), the core is interrupted when a single key press or multiple key presses are detected, depending on the interrupt mode chosen. In the

interrupt service routine for the keypad peripheral, the user can choose to read the KPAD_PRESSED bit of the KPAD_STAT register in periodic intervals to determine the number of times the key was meant to be pressed. During this state, all other key presses are ignored by the keypad interface. Once the key is released, the interface clears the KPAD_PRESSED bit. The KPAD_PRESSED bit indicates the state of the pressed key after the evaluation phase has ended.

Functional Description

The state diagram section describes the 16-pin programmable keypad interface.

State Diagram

The illustration shown in Figure 30-2 shows the different states of the keypad module. Once the KPAD_EN bit in the KPAD_CTL register is set, the keypad module goes into the Scan_Inputs state. In this state, all column outputs are driven with a value of 0 and the inputs are constantly read. If a key is pressed, it pulls down the corresponding row line which is read as 0. This event triggers the debounce counter and pushes the module into the Evaluate_Key_Pressed state.

In the Evaluate_Key_Pressed state, the state encoder drives a 0 on one column at a time and samples the input. If any of the inputs happen to be 0, then the inputs are sampled in a temporary register. This process is repeated for all valid columns (determined by the COLEN field of the KPAD_CTL register). Every time a 0 is observed on the row input, it is added with the previously added temporary register value. This is to register multiple keys pressed at the same time. Once all the columns are driven with one single 0 at a time, the interface moves the data in the temporary register to the KPAD_ROWCOL register. Once the data is sampled into the KPAD_ROWCOL register, its KPAD_ROW and KPAD_COL fields are checked for multiple 1s. If multiple 1s are found, then based on the KPAD_IRQMODE bits

in the KPAD_CTL register, an interrupt to the core is asserted. If no 1s are found, no interrupt is asserted. Next, the interface goes into the wait state where it checks for the pressed key to be released. Once the pressed key is released, it jumps to the Scan_Inputs state to detect the next key pressed. No matter which state the keypad is currently in, clearing the keypad enable bit of the KPAD_CTL register pushes the module into the disabled state.

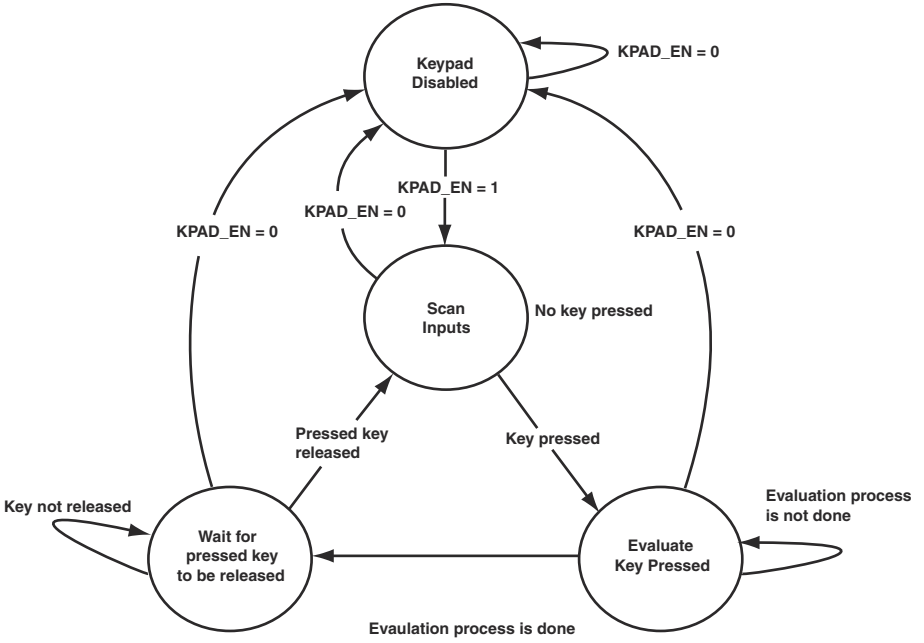


Figure 30-2. State Diagram

Programming Model

The following sections describe the programming model. The general procedure of programming the keypad module is:

1. Based on the characteristics of the keypad and application conditions, determine the column drive width and debounce time.

2. Use the actual SCLK and the formulae introduced in "KPAD_PRESCALE Register" and "KPAD_MSEL Register" subsections to calculate the register fields KPAD_PRESCALE_VAL, COLDRV_SCALE, and DBON_SCALE.

3. Write KPAD_PRESCALE and KPAD_MSEL registers based on the values obtained in step 2.

4. Write KPAD_CTL register to define the size of the keypad and the IRQ mode, and enable the keypad module.

5. In the interrupt service routine, read KPAD_STAT register to determine the status of the pressed key, and W1C to clear the bit KPAD_IRQ. Then read KPAD_ROWCOL to determine which key(s) is pressed, and then take application-specific actions accordingly.

Keypad Registers

Descriptions and bit diagrams for each of the memory-mapped registers (MMRs) are provided in the following subsections.

Table 30-2. Control/Status/Data Registers

Name	Address Offset	Access	Description
KPAD_CTL	0xFFC0 4100	R/W	"Keypad Control (KPAD_CTL) Register" on page 30-10
KPAD_PRESCALE	0xFFC0 4104	R/W	"Keypad Prescale (KPAD_PRESCALE) Register" on page 30-13
KPAD_MSEL	0xFFC0 4108	R/W	"Keypad Multiplier Select (KPAD_MSEL) Register" on page 30-15
KPAD_ROWCOL	0xFFC0 410C	R/WC	"Keypad Row-Column (KPAD_ROWCOL) Register" on page 30-15
KPAD_STAT	0xFFC0 4110	R/W1C	"Keypad Status (KPAD_STAT) Register" on page 30-18
KPAD_SOFTEVAL	0xFFC0 4114	R/W	"Keypad Software Evaluate (KPAD_SOFTEVAL) Register" on page 30-20

Keypad Control (KPAD_CTL) Register

The keypad control (KPAD_CTL) register, shown in Figure 30-3 and Table 30-3 is used to enable the keypad Interface module. This register programs the size of the input keypad matrix and interrupt modes, and controls the enabling/disabling of the Keypad module.

On reset, a read of this register returns a value of 0x0000, which implies that the keypad interface module is mapped onto a 1x1 keypad matrix. Reserved bits are read as 0s.

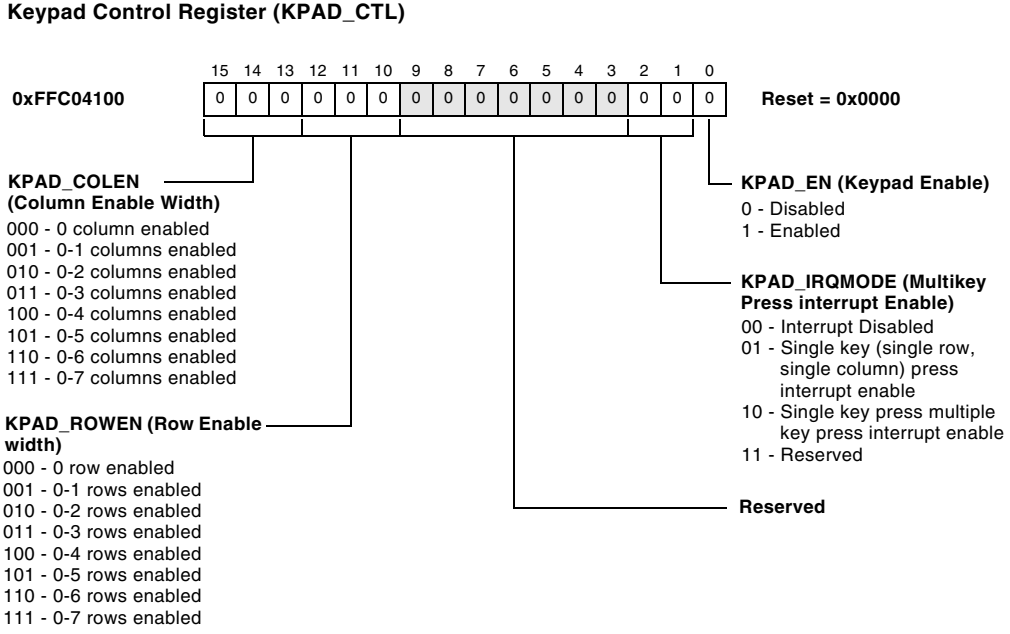


Figure 30-3. Keypad Control Register

Table 30-3. Keypad Control Register Bit Descriptions

Item	Bit(s)	Value	Description
KPAD_EN	0		Keypad enable bit.
		0	Disables the Keypad Interface module. Clearing this bit clears all the input enables, counters, drivers, resets the state of Keypad Interface module, and disables the device. When this bit is cleared, the Keypad Interface module loses the capability to generate interrupt request to the core.
		1	Enables the Keypad Interface module. Once this bit is enabled, the rest of the bits in this register are not allowed to change. When KPAD_EN bit is enabled, the only way to change any other bits is to clear KPAD_EN and reprogram rest of the bits.

Table 30-3. Keypad Control Register Bit Descriptions (Cont'd)

Item	Bit(s)	Value	Description
KPAD_IRQMODE	1-2		Multikey press interrupt enable bits. Enable the interrupt generation capability of the peripheral. These bits control the interrupt generation, based on the number of keys pressed simultaneously.
		b#00	Disables interrupts. Regardless of any input key of the keypad matrix being pressed, this causes the keypad interface module to lose the capability to generate interrupt requests to the core.
		b#01	A single key press generates an interrupt. Simultaneous multiple key presses do not generate an interrupt.
		b#10	A single key press (or multiple keys pressed in any row and column) generates an interrupt.
		b#11	Reserved. The keypad interface behavior becomes unpredictable if the user programs 11 into KPAD_IRQMODE.
Reserved	3-9		Reserved
KPAD_COLEN	13-15		Column enable width. This three-bit field programs the number of active columns. The value in this field + 1 determines the number of columns enabled in the input keypad matrix.
KPAD_ROWEN	10-12		Row enable width. This three-bit field programs the number of active rows. The value in this field + 1 determines the number of rows enabled in the input keypad matrix.

When keys are pressed in a single-row, multiple-column scenario (or a multiple-row, single-column scenario), it is possible to predict the pressed keys by reading the KPAD_ROWCOL register. When multiple keys in multiple rows (or multiple columns) are pressed simultaneously, it is not possible to predict the exact keys pressed with the existing hardware. The KPAD_MROWCOL bit of the KPAD_STAT register is used to make the distinction between the above two scenarios. It is up to the program to define the actions to take if it recognizes that multiple rows and multiple column keys are pressed simultaneously.

Examples

b#011 in KPAD_ROWEN and b#001 in KPAD_COLEN yields a 4-row, 2-column matrix.

b#010 in KPAD_ROWEN and b#001 in KPAD_COLEN yields a 3-row, 2-column matrix.

b#111 in KPAD_ROWEN and b#111 in KPAD_COLEN yields an 8-row, 8-column matrix.

Keypad Prescale (KPAD_PRESCALE) Register

The KPAD_PRESCALE register, shown in Figure 30-4, is used to program the pre-scale value that would be used in deriving delay parameters that the interface module should be sensitive to.

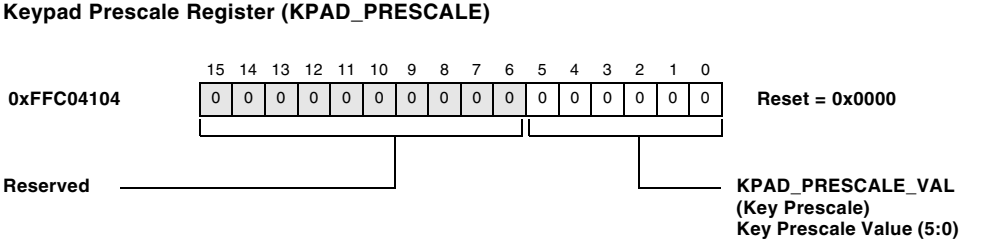


Figure 30-4. Key Prescale Register

The KPAD_PRESCALE is a 16-bit register. The lower 6 bits are programmable and the rest of the bits are reserved. This makes the dynamic range of the prescale = 1 - 64. The value in the prescale register is used to calculate both debounce period (Tdb) and column drive period (Tcw).

The KPAD_PRESCALE register is used to establish a convenient time base so that the user could use the KPAD_MSEL register to generate the necessary time delays.

The formula to get a timescale of T follows:

$$Prescale\ Value = \frac{\langle SCLK\ Frequency \rangle \times T}{1024} - 1$$

When using this formula, Note the unit of SCLK and the time scale must agree. For example, to generate a 0.1 ms timescale, the formula is:

$$Prescale\ Value = \frac{\langle SCLK\ Frequency\ in\ MHz \rangle \times 100ms}{1024} - 1$$

Taking some real numbers of SCLK:

SCLK = 133 MHz, to generate 0.1 ms time base, KPAD_PRESCALE[5:0] = 12

SCLK = 50 MHz, to generate 0.1 ms time base, KPAD_PRESCALE[5:0] = 4

SCLK = 10 MHz, to generate 0.1 ms time base, KPAD_PRESCALE[5:0] = 0

This register cannot be written once the keypad is enabled.

Keypad Multiplier Select (KPAD_MSEL) Register

The KPAD_MSEL register, shown in Figure 30-5, is used to program different delay parameters (column drive width Tcw and debounce time Tdb) that the keypad module should be sensitive to.

The settings (COLDRV_SCALE, DBON_SCALE) of KPAD_MSEL register are determined by the values of Tcw and Tdb. They can be calculated as follows:

```
COLDRV_SCALE = [(Tcw * SCLK)/ (KPAD_PRESCALE +1) * 1024)]-1

DBON_SCALE = [(Tdb * SCLK)/ (KPAD_PRESCALE +1) * 1024)]-1
```

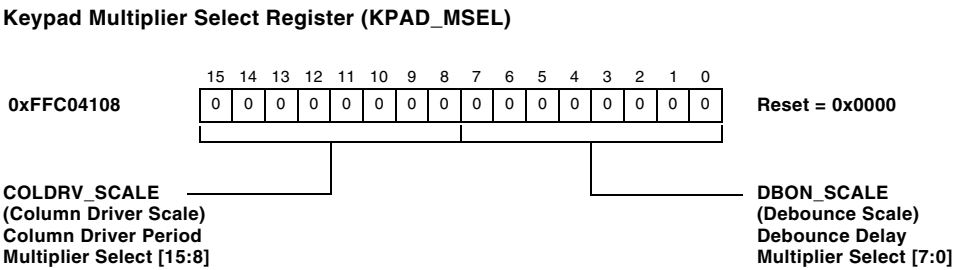


Figure 30-5. Keypad Multiplier Select Register

Keypad Row-Column (KPAD_ROWCOL) Register

The KPAD_ROWCOL register, shown in Figure 30-6, is used to register the input row values and column output values once the interface logic gets to a valid state.

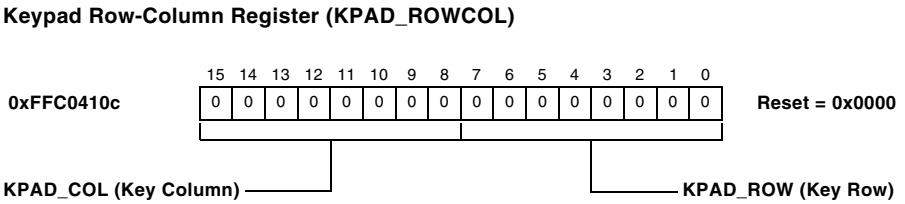


Figure 30-6. Keypad Row-Column Register

The KPAD_ROWCOL register is used to determine the keys pressed. In the Evaluate_Key_Pressed state, each column is driven with a low value and the rest of the columns are tri-stated. Row inputs are read and, if at least one input is found to be zero, the corresponding row inputs and column outputs are accumulated in the temporary register. This process is repeated for all of the columns, one at a time. Once all of the columns are individually driven with a low value, then the interface moves data contents in the temporary register into the KPAD_ROWCOL register. By the end of the evaluation state, the KPAD_ROWCOL register has information about whether a single key is pressed, multiple keys have been pressed, or no key is pressed. Based on the values of the KPAD_IRQMODE bits in the KPAD_CTL register and the number of keys pressed, an interrupt to the core is asserted. A value of 1 in KPAD_ROW implies that a key is pressed, and a value of 0 implies that a key has not been pressed.

A write to the KPAD_ROWCOL register clears the register (loads with a value of 0x0000). A read of this register on reset returns a value of 0x0000.

Figure 30-7 provides an explanation of the KPAD_ROWCOL register and interrupt generation when key x is pressed. Case 6 shows the situation where it is not possible to distinguish the keys when they are pressed in multiple rows and columns.

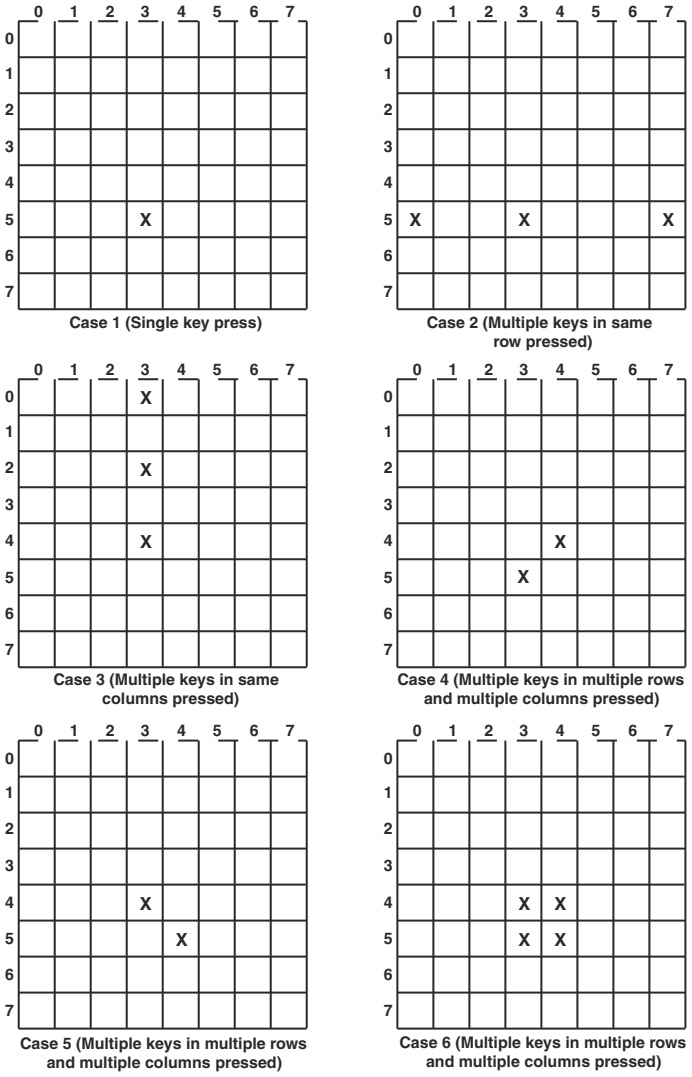


Figure 30-7. Interrupt Generation When X-Key Pressed

Keypad Status (KPAD_STAT) Register

The KPAD_STAT register, shown in Figure 30-8, is used to hold and clear the status of the interrupt generated by the keypad module. It is also helpful in resolving the pressed keys when multiple keys are pressed simultaneously.

The KPAD_IRQ bit in the KPAD_STAT register is used to indicate that there is an interrupt request generated by the Keypad Interface module. The KPAD_IRQMODE bits in KPAD_CTL are the interrupt enable bits of the keypad interface. If KPAD_IRQMODE = b#00, the peripheral loses the capability to generate an interrupt. The KPAD_IRQ is asserted once the module evaluates the keys pressed, based on the KPAD_IRQMODE bits and the number of keys pressed. Assertion of this bit signifies that an interrupt request to the core is asserted. This bit is a sticky bit, which means that, once asserted, it remains asserted until the user clears it. This bit is cleared on reset, when the KPAD_EN bit in the KPAD_CTL register is cleared or by writing a 1 to the KPAD_IRQ bit in the KPAD_STAT register.

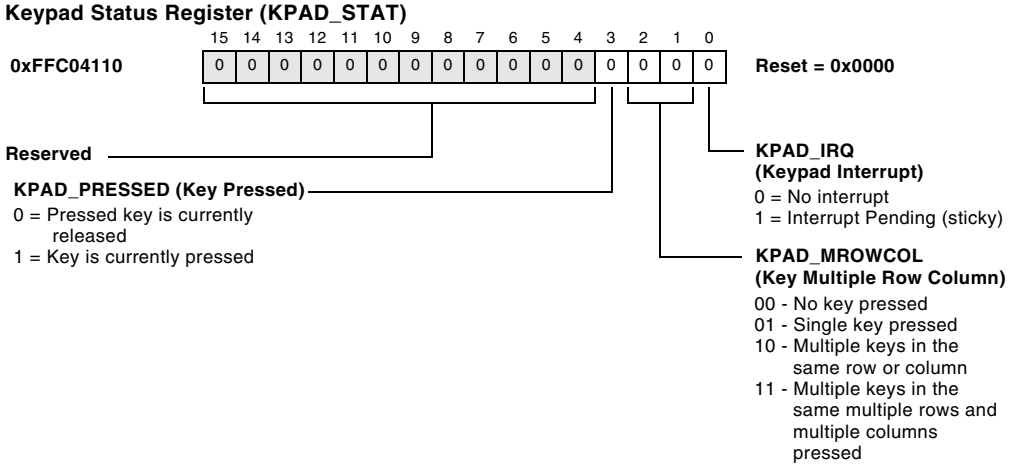


Figure 30-8. Keypad Status Register

The KPAD_MROWCOL bits are used to indicate whether multiple rows and columns in the KPAD_ROWCOL register are asserted at the same time. Physically, this would mean that more than one row key and more than one column key have been pressed simultaneously. In this scenario, it becomes impossible to predict the exact keys pressed with the existing hardware. The user could choose to ignore the key press action. These bits become particularly handy when one row and multiple columns or one column and multiple rows of the KPAD_ROWCOL register are pressed simultaneously. In these scenarios it is possible to detect the exact keys pressed by reading the KPAD_ROWCOL register. Interrupt generation in this situation is enabled by KPAD_IRQMODE bits in the KPAD_CTL register. In the interrupt service routine the user can read the status of the KPAD_MROWCOL bits in the KPAD_STAT register to determine multiple row and multiple column keys have been pressed or not and appropriate action can be taken. These bits get number of keys pressed information from the KPAD_ROWCOL register. These bits are cleared when the KPAD_ROWCOL register is cleared. The KPAD_ROWCOL register is cleared by doing a PAB write to the KEY_ROWCOL register.

The KPAD_PRESSED bit indicates the state of the pressed key after the evaluation phase has ended. This bit remains high until the pressed key is released. This bit could be used by the customer to implement the Press-Hold feature. Once the key is pressed, the keypad interface generates an interrupt provided the KPAD_IRQMODE bits in the KPAD_CTL register are set appropriately. The user could choose to read the KPAD_PRESSED bit of the status register in the interrupt service retinue to determine if the pressed key is released or not and then take the appropriate action. This bit is cleared once the interface gets into the scan inputs state. On reset, a read of this register returns a value of 0x0000.

Keypad Software Evaluate (KPAD_SOFTEVAL) Register

The KPAD_SOFTEVAL register, shown in Figure 30-9, is used to force the interface into the evaluate state.

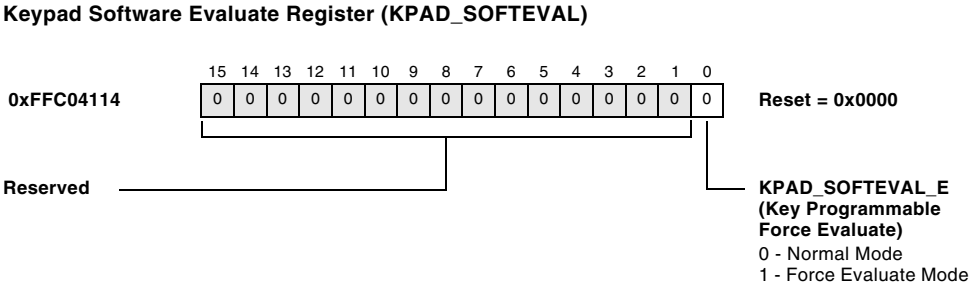


Figure 30-9. Keypad Software Evaluate Register

Enabling the SOFTEVAL bit forces the interface to go through the evaluate phase. If the interface is either in the Scan_Inputs state or Wait state, a write to the SOFTEVAL bit in the KPAD_SOFTEVAL register causes the interface to jump to the evaluate phase. At the end of the evaluation phase, an interrupt to the core is asserted based on the value in the KPAD_ROWCOL register and KPAD_IRQMODE. The SOFTEVAL bit is cleared at the end of the evaluate phase. If the write to the SOFTEVAL bit happens when the module is in the evaluation phase, the interface proceeds with its normal sequence of actions except that at the end of the evaluation phase, the SOFTEVAL bit is cleared.

Programming Examples

Listing 30-1 describes the configuration of the keypad module.

Listing 30-1. Configure Keypad Module

```
/* Configure the prescale register KPAD_PRESCALE */
P0.L = LO(KPAD_PRESCALE);
P0.H = HI(KPAD_PRESCALE);
R0.L = 4;    /*0.1 MS WITH 50MHz SCLK */
W[P0] = R0.L;
/* Configure the column drive width and debounce time */
P0.L = LO(KPAD_MSEL);
P0.H = HI(KPAD_MSEL);
R0.L = 0x0909   /* 1 MS WITH 50MHz SCLK, for both column drive
width and debounce time */
W[P0] = R0.L;
/* Configure the KPAD_CTL register to set keypad size to 5 rows
by 6 columns, multiple key press interrupt enabled, keypad module
enabled */
P0.L = LO(KPAD_CTL);
P0.H = HI(KPAD_CTL);
R0.L = 0xB005;
W[P0] = R0.L;
SSYNC;
```

31 CAN MODULE

This chapter describes the controller area network (CAN) modules. Familiarity with the CAN standard is assumed. Refer to Version 2.0 of *CAN Specification* from Robert Bosch GmbH.

This chapter includes the following sections:

- "Overview" on page 31-1
- "Interface Overview" on page 31-2
- "CAN Operation" on page 31-10
- "Functional Operation" on page 31-24
- "CAN Registers" on page 31-41
- "Programming Examples" on page 31-85

Overview

The ADSP-BF544, ADSP-BF548, and ADSP-BF549 Blackfin processors have two separate and identical CAN modules, referred to as CAN0 and CAN1. The CAN1 module is not present on ADSP-BF542 derivatives. There are no CAN modules present on the ADSP-BF547 Blackfin processor. Neither of the CAN modules may be available on commercial and/or industrial grade products. For more information see *ADSP-BF542/544/547/548/549 Embedded Processor Data Sheet*.

Key features of the CAN module include:

- Conformity to the CAN 2.0B (active) standard
- Support for standard (11-bit) and extended (29-bit) identifiers
- Support for 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 31-1 for a symbolic representation of the CAN transceiver interconnection, and Figure 31-2 for a block diagram. The Blackfin processor's `CANxTX` output and `CANxRX` input pins are connected to an external CAN transceiver's TX and RX pins (respectively). The `CANxTX` and `CANxRX` pins operate with TTL levels and are appropriate for operation with CAN bus transceivers according to ISO/DIS 11898.

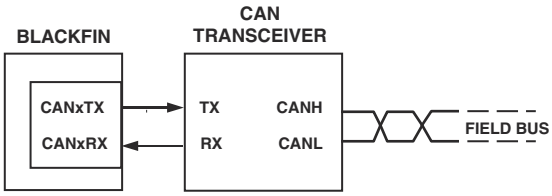


Figure 31-1. Representation of CAN Transceiver Interconnection

The CANxRX and CANxTX signals can be found on GPIO Port G, pins PG12–PG15. By default, these pins are in GPIO mode. To enable CAN functionality, the appropriate bits must be set in the PORTG_FER register. If CAN0 is used, set bits 12 and 13. If CAN1 is used, set bits 14 and 15.

Additionally, the associated bit fields of the PORTG_MUX register must be kept zero, which is their default value. CAN data is defined to be either *dominant* (logic 0) or *recessive* (logic 1). The default state of the CANxTX output is recessive.

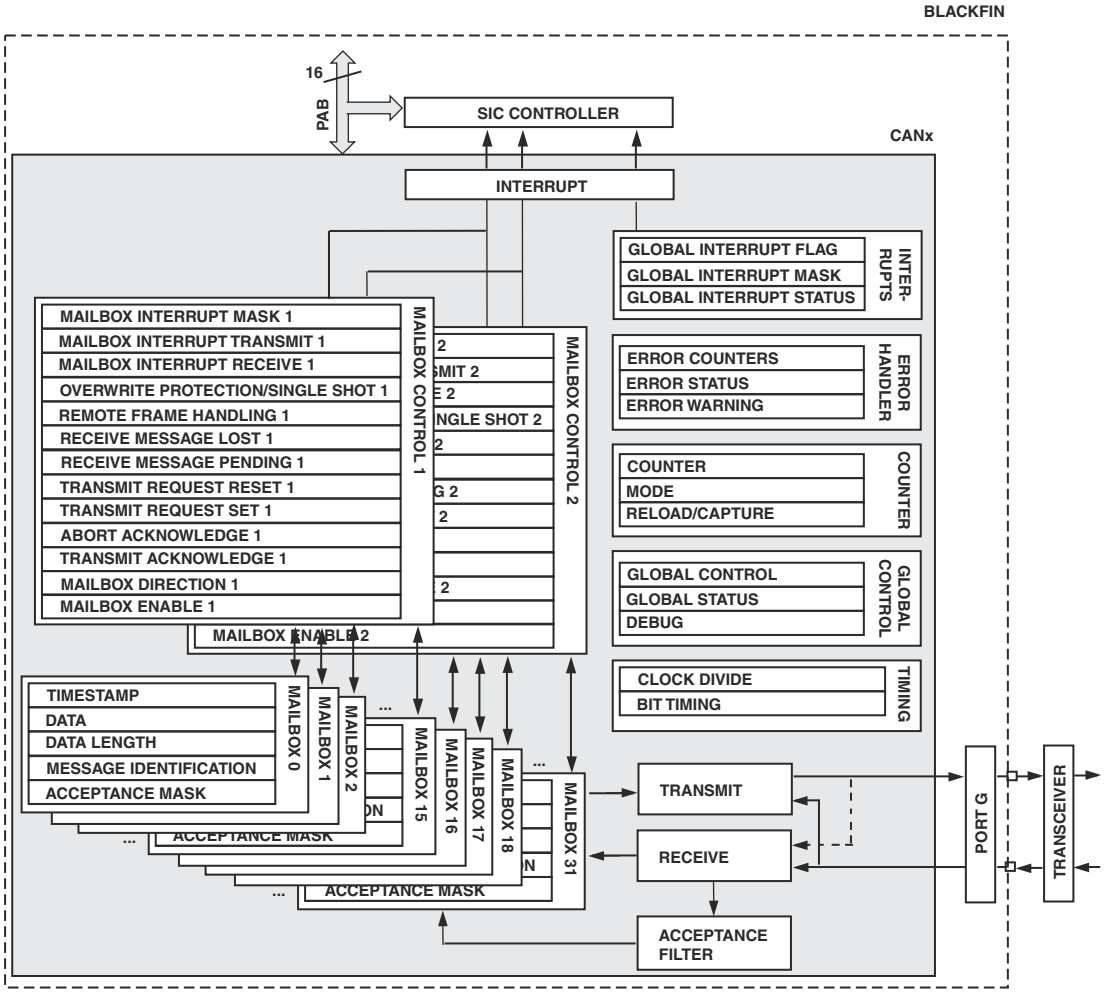


Figure 31-2. CANx Block Diagram

The PG13 pin (CAN0RX input pin) is also internally routed to the alternate capture input TACI4 of the GP timer 4. Similarly, the PG15 pin (CAN1RX input pin) also goes to the alternate capture input TACI5 of the GP timer 5. This way, GP timers 4 and 5 can be used to auto-detect or adjust the bit rate on the CAN bus.

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. See Figure 31-3.

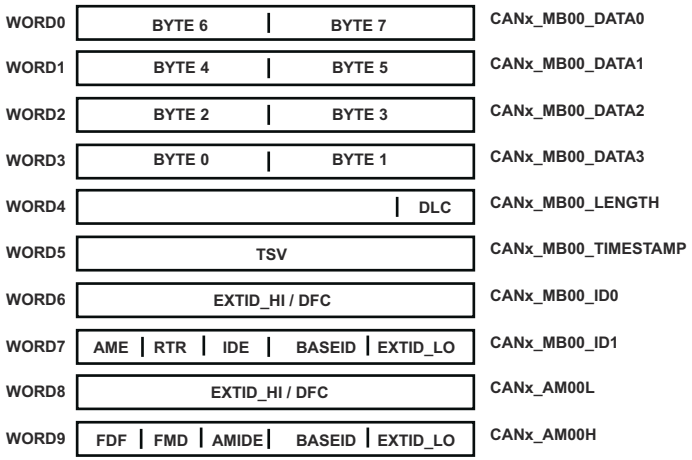


Figure 31-3. CAN Mailbox Area

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.

The CAN mailbox identification (CANx_MBxx_IDO/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 CANx_MCx is set).

The other mailbox area registers are:

- The data length code (DLC) in CANx_MBxx_LENGTH. The upper 12 bits of CANx_MBxx_LENGTH of each mailbox are marked as reserved. These 12 bits should always be set to 0.
- Up to eight bytes for the data field, sent MSB first from the CANx_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 CANx_MBxx_DATA3 register.
- Two bytes for the time stamp value (TSV) in the CANx_MBxx_TIMESTAMP register

The final registers in the mailbox area are the acceptance mask registers (CANx_AMxxH and CANx_AMxxL). The acceptance mask is enabled when the AME bit is set in the CANx_MBxx_ID1 register. If the “filtering on data field” option is enabled (DNM = 1 in the CANx_CONTROL register and FDF = 1 in the corresponding acceptance mask), the EXTID_HI[15:0] bits of CANx_MBxx_IDO are reused as acceptance code (DFC) for the data field filtering. For more information, see “Receive Operation” on page 31-16.

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 (`CANx_MDx`) would control mailboxes as shown in Figure 31-4.

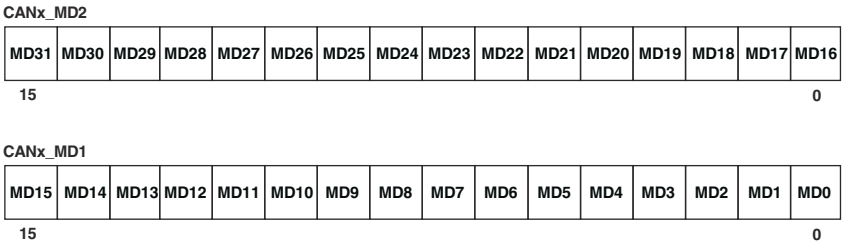


Figure 31-4. CAN Register Pairs

The mailbox control register area consists of these register pairs:

- `CANx_MC1` and `CANx_MC2` (mailbox enable registers)
- `CANx_MD1` and `CANx_MD2` (mailbox direction registers)
- `CANx_TA1` and `CANx_TA2` (transmit acknowledge registers)
- `CANx_AA1` and `CANx_AA2` (abort acknowledge registers)
- `CANx_TRS1` and `CANx_TRS2` (transmit request set registers)
- `CANx_TRR1` and `CANx_TRR2` (transmit request reset registers)
- `CANx_RMP1` and `CANx_RMP2` (receive message pending registers)
- `CANx_RML1` and `CANx_RML2` (receive message lost registers)

- CANx_RFH1 and CANx_RFH2 (remote frame handling registers)
- CANx_OPSS1 and CANx_OPSS2 (overwrite protection/single shot transmission registers)
- CANx_MBIM1 and CANx_MBIM2 (mailbox interrupt mask registers)
- CANx_MBTIF1 and CANx_MBTIF2 (mailbox transmit interrupt flag registers)
- CANx_MBRIF1 and CANx_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 CANxRX and CANxTX pins are TTL-compliant signals, the CAN signals beyond the transceiver (see Figure 31-1 on page 31-3) have asymmetric drivers. A low state on the CANxTX 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 CANxTX 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 CANxRX pin to verify that the logic level being driven is the value it just placed on the CANxTX pin. This is where the names for the logic levels apply. If a transmitting node places a recessive '1' on CANxTX and detects a dominant

‘0’ on the CANxRX 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 CANxRX is the value driven on CANxTX, 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 31-5 for more details regarding CAN frame structure.

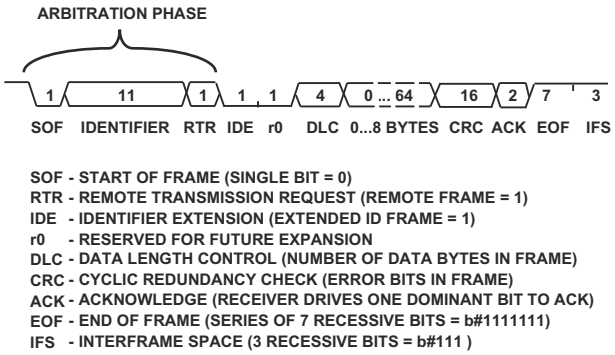


Figure 31-5. Standard CAN Frame

Figure 31-5 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 31-6.

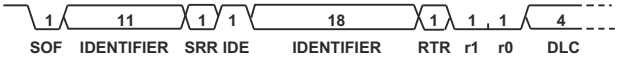


Figure 31-6. Extended CAN Frame

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 CANx_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 CANx_CLOCK register defines the TQ value, and multiple time quanta make up the duration of a CAN bit on the bus. The CANx_TIMING register controls the nominal bit time and the sample point of the individual bits

in the CAN protocol. Figure 31-7 shows the three phases of a CAN bit—the synchronization segment, the segment before the sample point, and the segment after the sample point.

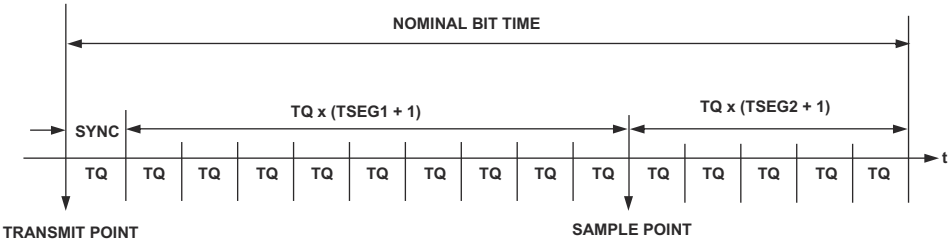


Figure 31-7. Three Phases of a CAN Bit

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 CANx_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 t_{BIT} = 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 recessive-to-dominant edge outside the synchronization segment, it can automatically move the sampling point such that the CAN bit is still handled properly. The synchronization jump

width (SJW) field specifies the maximum number of TQs, ranging from 1 to 4 (SJW + 1), allowed for such a re-synchronization attempt. The SJW value should not exceed TSEG2 or TSEG1. Therefore, the fundamental rule for writing CANx_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 CANxRX 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 CANx_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 CANx_CLOCK or CANx_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 of processor reset or hibernate, enter configuration mode by setting the CCR bit in the master control (CANx_CONTROL) register and poll the global CAN status (CANx_STATUS) register until the CCA bit is set.

If the GPIO pins are not first enabled for CAN functionality, the module does not enter configuration mode.

If the TSEG1 field of the CANx_TIMING register is programmed to ‘0,’ the module does not leave configuration mode.

During configuration mode, the module is not active on the CAN bus line. The CANxTX 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 soft reset does not change the values of CANx_CLOCK and CANx_TIMING. Thus, an ongoing transfer through the CAN bus cannot be corrupted by changing the bit timing parameter or initiating the soft reset (SRS = 1 in CANx_CONTROL).

Transmit Operation

Figure 31-8 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 CANx_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 CANx_MCx) and, subsequently, the corresponding transmit request bit is set (TRSn = 1 in CANx_TRSx).

When a transmission completes, the corresponding bits in the transmit request set register and in the transmit request reset register (TRRn in CANx_TRRx) are cleared. If transmission was successful, the corresponding bit in the transmit acknowledge register (TAn in CANx_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 CANx_AAx) is set. A requested transmission can also be manually aborted by setting the corresponding TRRn bit in CANx_TRRx.

Multiple CANx_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 CANx_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 CANx_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 31-8). The currently aborted transmission is restarted after any messages with higher priority are sent.

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message 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.

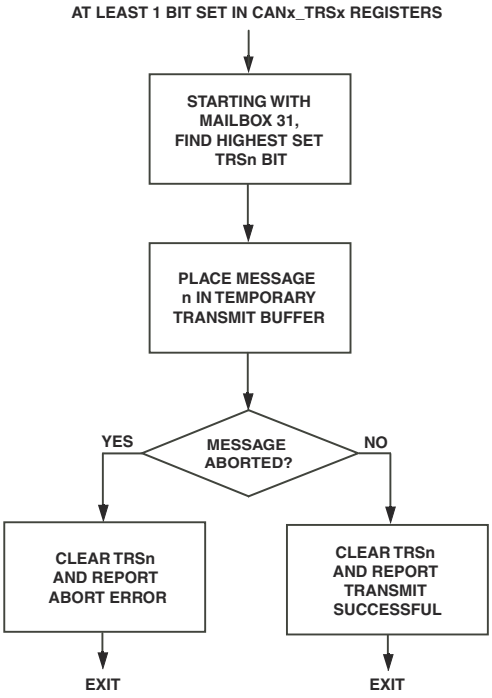


Figure 31-8. CAN Transmit Operation Flow Chart

Single Shot Transmission

If the single shot transmission feature is used (`OPSSn` = 1 in `CANx_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 `CANx_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 CANx_UCRC register. When enabled in this mode (set UCCNF[3:0] = 0x3 in CANx_UCCNF), the counter (CANx_UCCNT) is loaded with the value in the CANx_UCRC register. The counter decrements at the CAN bit clock rate down to 0 and is then reloaded from CANx_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 is 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 CANx_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. In standard mode, the message is compared to the content of the CANx_MByy_ID1 register. In extended mode, the content of the CANx_MByy_ID0 register must also match.

If the AME bit is not set, a match is signalled only if IDE, RTR, and all (11 or 29) 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.

The following logic applies:

- (Received Message ID XNOR CANx_MBxx_ID0/1)

  or

- (AME and CANx_AMxxH/L).

This logic appears graphically in Figure 31-9.

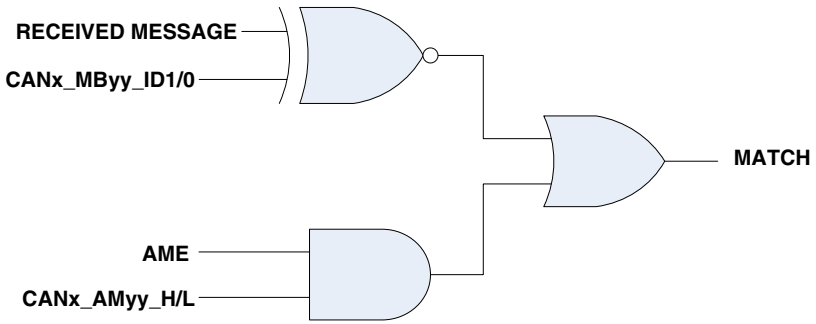


Figure 31-9. CAN Receive Message Logic

A one at the respective bit position in the CAN_AMxxH/L mask registers means that the bit does not need to match when AME=1. This way, a mailbox can accept a group of messages.

Table 31-1. 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 31-10 illustrates the decision tree of the receive logic when processing the individual mailboxes.

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 `CANx_RMLx`) is set and the stored message is overwritten. This results in the receive message lost interrupt being raised in the global CAN interrupt status register (`RMLIS` = 1 in `CANx_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.

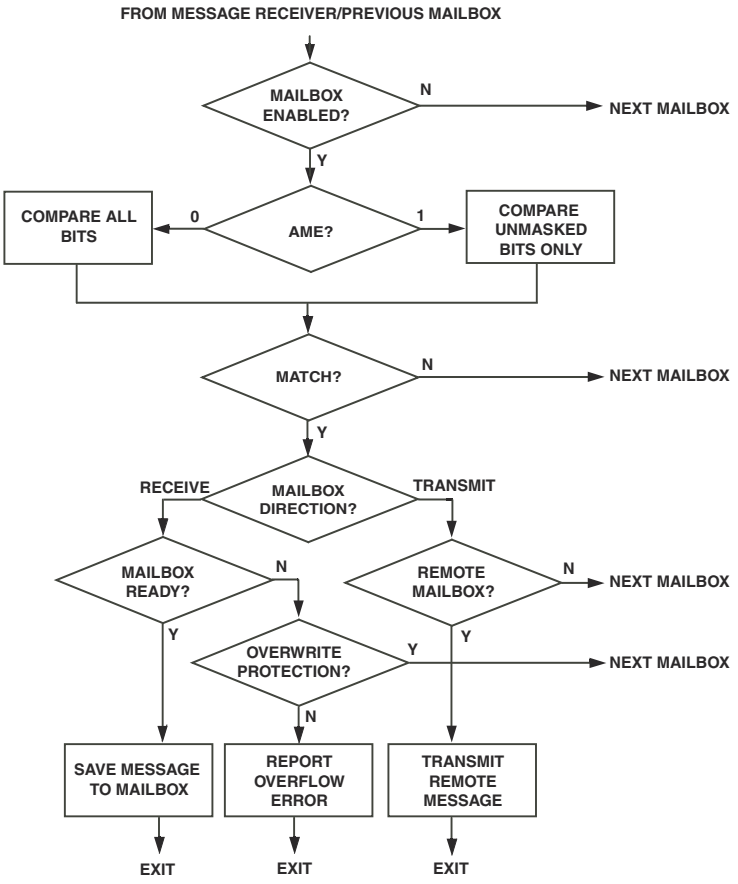


Figure 31-10. CAN Receive Operation Flow Chart

Data Acceptance Filter

If DeviceNet mode is enabled (DNM = 1 in CANx_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 31-2.

Table 31-2. 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 CANx_AMxxH register, the CANx_AMxxL register holds the data field mask (DFM[15:0]). If the FDF bit is cleared in the corresponding CANx_AMxxH register, the CANx_AMxxL register holds the extended identifier mask (EXTID_HI[15:0]).

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 CANx_UCCNF), the counter in the CANx_UCCNT register is loaded with the predefined value contained in the CAN universal counter reload/capture register (CANx_UCRC). This counter then decrements at the CAN bit rate. If the UCCT and UCRC bits in the CANx_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 CANx_UCRC contents. If the counter has counted down to 0 without receiving a message in mailbox 4, the UCEIS bit in the global CAN interrupt status (CANx_GIS) register is set, and the counter is automatically reloaded with the contents of the CANx_UCRC

register. If an interrupt is desired, the UCEIM bit in the CANx_GIM register must also be set. With the mask bit set, when a watchdog interrupt occurs, the UCEIF bit in the CANx_GIF register is also set.

The counter can be reloaded with the contents of CANx_UCRC or disabled by writing to the CANx_UCCNF register.

The time period it takes for the watchdog interrupt to occur is controlled by the value written into the CANx_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 CANx_UCCNF). The value of the 16-bit free-running counter (CANx_UCCNT) is then written into the CANx_MBxx_TIMESTAMP register of the corresponding mailbox when a received message is stored or a message is 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 CANx_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 CANx_UCCNF register. The counter can also be loaded with a value by writing to the counter register itself (CANx_UCCNT).

It is also possible to clear the counter (CANx_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 CANx_UCCNF register.

An overflow of the counter sets a bit in the global CAN interrupt status register (UCEIS in the CANx_GIS register). A global CAN interrupt can optionally occur by unmasking the bit in the global CAN interrupt mask register (UCEIM in the CANx_GIM register). If the interrupt source is unmasked, a bit in the global CAN interrupt flag register is also set (UCEIF in the CANx_GIF register).

Remote Frame Handling

Automatic handling of remote frames can be enabled for a transmit mailbox by setting the corresponding bit in the remote frame handling registers (CANx_RFHx) of a transmit mailbox.

Remote frames are data frames with no data field and the RTR bit set. The data length code of the responding data frame is overruled by the DLC of the requesting 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 mailboxes can receive and transmit remote frame requests. When setup for automatic remote frame handling, the CANx_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.

To avoid the risk of inconsistent messages, it is recommended to temporarily disable the mailbox while its data registers are updated. See "Temporarily Disabling Mailboxes".

Temporarily Disabling Mailboxes

If a mailbox is enabled and configured as "transmit," write accesses to the data field should be guarded to avoid transmission of inconsistent messages. Special care must be taken if the mailbox is transmitting (or attempting to transmit) repeatedly. Also, 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 (CANx_MBTD).

The pointer to the requested mailbox must be written to the TDPTR[4:0] bits of the CANx_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 CAN 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 CANx_MBIMx.

If a mailbox is configured as a receive mailbox, the corresponding receive interrupt flag is set (MBRIFn = 1 in CANx_MBRIFx) after a received message is stored in mailbox n (RMPn = 1 in CANx_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 CANx_MBRIFx, the MBRIRQ interrupt output is raised in CANx_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 CANx_MBRIFx. Prior to this, the RMPn bit must also be cleared by software.

If a mailbox is configured as a transmit mailbox, the corresponding transmit interrupt flag is set (MBTIFn = 1 in CANx_MBTIFx) after the message in mailbox n is sent correctly (TAn = 1 in CANx_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 CANx_MBTIFx, the MBTIRQ interrupt output is raised in CANx_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 CANx_MBTIFx. Additionally, software must clear the associated TAn bit or set the associated TRSn bit to clear the interrupt source that asserts the MBTIFn bit.

Global CAN Interrupt

The global CAN interrupt logic is implemented with three registers—the global CAN interrupt mask register (CANx_GIM), where each interrupt source can be enabled or disabled separately; the global CAN interrupt status register (CANx_GIS); and the global CAN interrupt flag register (CANx_GIF). The interrupt mask bits only affect the content of the global CAN interrupt flag register (CANx_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 CAN 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 CAN interrupt output (GIRQ) bit in the global CAN interrupt status register is only asserted if a bit in the CANx_GIF register is set. The GIRQ bit remains set as long as at least one bit in the interrupt flag register CANx_GIF is set. All bits in the interrupt status and in the interrupt flag registers remain set until cleared by software or a soft reset has occurred.

In the ISR, the interrupt latch should be cleared by a W1C operation to the corresponding bit of the CANx_GIS register. This clears the related bits of both the CANx_GIS and CANx_GIF registers.

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.
- **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 is received for a mailbox that currently contains unread data. At least one bit in the receive message lost register (CANx_RMLx) is set. If the bit in CANx_GIS (and CANx_GIF) is reset and there is at least one bit in CANx_RMLx still set, the bit in CANx_GIS (and CANx_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CANx_RMLx is set.
- **Abort acknowledge interrupt** (AAIM, AAIS, AAIF)
  At least one AAn bit in the abort acknowledge registers CANx_AAx is set. If the bit in CANx_GIS (and CANx_GIF) is reset and there is at least one bit in CANx_AAx still set, the bit in CANx_GIS (and CANx_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_GIS (and CANx_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 CANx_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 CANx_CEC register, the CANx_ESR register, and the CANx_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 (CANx_EWR) register. After powerup reset, the CANx_EWR register is set to the default warning level of 96 for both error counters. After soft 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 (CANxRX) 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 is sent successfully), a bit error is signaled any time the value on CANxRX does not equal what is being transmitted on CANxTX.
- **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 is 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 re-synchronize 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 CANx_ESR as well as the error counter register CANx_CEC. In addition to the standard errors, the CANx_ESR register features a flag that signals when the CANxRX 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 31-11.

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.

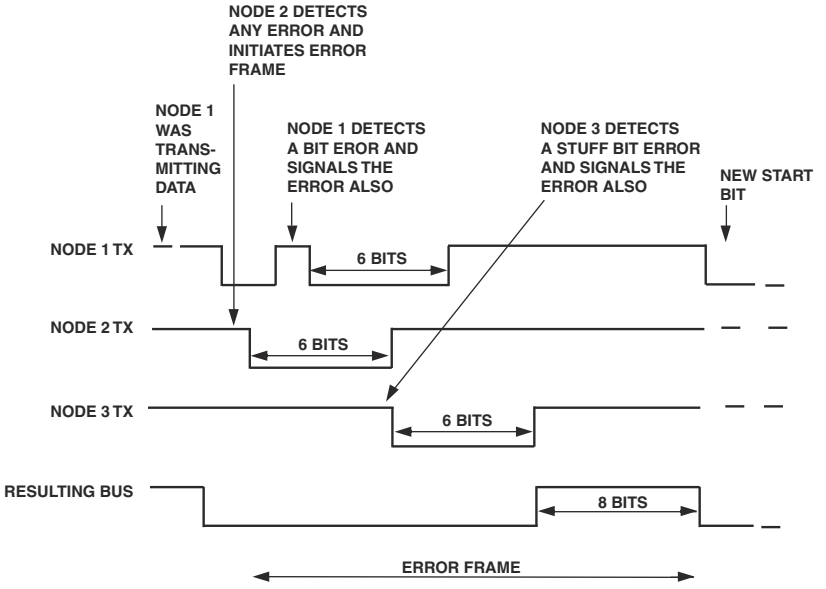


Figure 31-11. CAN Error Scenario Example

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 is 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 31-3. 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 CANx_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 CANx_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 CANx_STATUS. Software intervention is required to recover from this state, unless the ABO bit in the CANx_CONTROL register is enabled.

Table 31-3. 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 CANx_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, CANx_EWR. More information is available on page 31-85.

Additionally, interrupts can occur for all of these levels by unmasking them in the global CAN interrupt mask register (CANx_GIM) shown on page 31-49. 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 CANx_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 CANx_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 31-1 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 CANx_STATUS register) or in suspend mode (CSA = 1 in CANx_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 CANx_DEBUG register. When the CDE bit is cleared, all debug features are disabled.

Listing 31-1. Enabling CAN0 Debug Features in C on the ADSP-BF549

```
#include <cdefBF549.h>
/* Enable debug mode, CDE must be set before other flags can be
changed in register */
*pCAN0_DEBUG |= CDE ;

/* Set debug flags */
*pCAN0_DEBUG &= ~DTO ;
*pCAN0_DEBUG |= MRB | MAA | DIL ;

/* Run test code */

/* Disable debug mode */
*pCAN0_DEBUG &= ~CDE ;
```

When the CDE bit is set, it enables writes to the other bits of the CANx_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 CANx_CLOCK and CANx_TIMING registers.
- Allows write access to the read-only transmit/receive error counter register CANx_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 through 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 CANxTX 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 CANxTX output pin. When this bit is set, the CANxTX pin continuously drives recessive bits.

The disable receive input bit (DRI) is used to disable the CANxRX 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 CANxRX input pin is ignored.

The disable error counters bit (DEC) is used to disable the transmit and receive error counters in the CANx_CEC register. When this bit is set, the CANx_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 31-4 shows several common combinations of test mode bits.

Table 31-4. 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. CANxRX input is enabled.

Table 31-4. CAN Test Modes (Cont'd)

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
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. CANxRX 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. CANxRX input is ignored. Internal loop is enabled
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 CANxTX. CANxRX 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 CANx_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 CANx_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 CANxTX 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 CANx_CONTROL. The only differences between suspend mode and configuration mode are that writes to the CANx_CLOCK and CANx_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 CANx_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 CANx_INTR. When the CAN module is in sleep mode, all register reads return the contents of CANx_INTR instead of the usual contents. All register writes, except to CANx_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 CANx_INTR register ends sleep mode. If the WBA bit in the CANx_CONTROL register is set before entering sleep mode, a dominant bit on the CANxRX pin also ends sleep mode.

When software sets the SMR bit, hardware sets the CSR bit as well, making sleep mode a super set of suspend mode. When the controller wakes up from sleep mode, hardware automatically clears the SMR and CSR bits. If, however, the controller never enters sleep mode, because the wake-up condition was met before the SMACK bit turns to one, the SMR and CSR may not get cleared automatically. Therefore, it is good programming practice to always clear the SMR and CSR by software when returning from 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 through 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 CANxRX pin. If the transceiver then senses CAN bus activity, it will, in turn, drive the CANxRX pin to the dominant logic '0' level. This signals to the Blackfin processor that CAN bus activity is 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 CANxRX 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 b#00.

Soft Reset

The CAN controller features a build-in reset mechanism which is referred to as soft reset. Soft reset is entered immediately after software has set the SRS bit in the CANx_CONTROL register. Soft reset brings all control registers to a defined state. Mailbox and error registers remain unaffected. Soft reset does not alter the CANx_TIMING and CANx_CLOCK registers and does not disturb the on-going transmission of a currently pending message, acknowledge bit or error frame. However, when recovering from soft reset, software may lose track of transmission or reception reports and interrupts.

CAN Registers

The following sections describe the CAN controller register definitions.

Table 31-5 through Table 31-9 show the functions of the CAN controller registers.

Table 31-5. CAN Global Registers

Register Name	Description	Notes
CANx_CONTROL	"Master Control (CANx_CONTROL) Registers" on page 31-45	Reserved bits 15:8 and 3 must always be written as '0'
CANx_STATUS	"Global CAN Status (CANx_STATUS) Registers" on page 31-46	Write accesses have no effect
CANx_DEBUG	"CAN Debug (CANx_DEBUG) Registers" on page 31-47	Use of these modes is not CAN-compliant
CANx_CLOCK	"CAN Clock (CANx_CLOCK) Registers" on page 31-47	Accessible only in configuration mode
CANx_TIMING	"CAN Timing (CANx_TIMING) Registers" on page 31-48	Accessible only in configuration mode
CANx_INTR	"CAN Interrupt (CANx_INTR) Registers" on page 31-48	Reserved bits 15:8 and 5:4 must always be written as '0'
CANx_GIM	"Global CAN Interrupt Mask (CANx_GIM) Registers" on page 31-49	Bits 15:11 and 9 are reserved
CANx_GIS	"Global CAN Interrupt Status (CANx_GIS) Registers" on page 31-49	Bits 15:11 and 9 are reserved
CANx_GIF	"Global CAN Interrupt Flag (CANx_GIF) Registers" on page 31-50	Bits 15:11 and 9 are reserved

Table 31-6. CAN Mailbox/Mask Registers

Register Name	Description	Notes
CANx_AMxxH CANx_AMxxL	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50	Do not write when mailbox MBxx is enabled
CANx_MBxx_ID1 CANx_MBxx_ID0	"Mailbox Word 7 (CANx_MBxx_ID1) Registers" on page 31-54	Do not write when mailbox MBxx is enabled
CANx_MBxx_TIMESTAMP	"Mailbox Word 5 (CANx_MBxx_TIMESTAMP) Registers" on page 31-58	Holds timestamp information when timestamp mode is active

Table 31-6. CAN Mailbox/Mask Registers (Cont'd)

Register Name	Description	Notes
CANx_MBxx_LENGTH	"Mailbox Word 4 (CANx_MBxx_LENGTH) Registers" on page 31-59	Values greater than 8 are not allowed. Bits 15:4 are reserved.
CANx_MBxx_DATA3 CANx_MBxx_DATA2 CANx_MBxx_DATA1 CANx_MBxx_DATA0	"Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers" on page 31-61	Software controls reading correct data, based on DLC

Table 31-7. CAN Mailbox Control Registers

Register Name	Description	Notes
CANx_MC1 CANx_MC2	"Mailbox Configuration (CANx_MCx) Registers" on page 31-69	Always disable before modifying mailbox area or direction
CANx_MD1 CANx_MD2	"Mailbox Direction (CANx_MDx) Registers" on page 31-70	Never change MDn direction when mailbox n is enabled. MD[31:24] and MD[7:0] are read only
CANx_RMP1 CANx_RMP2	"Receive Message Pending (CANx_RMPx) Registers" on page 31-71	Clearing RMPn bits also clears corresponding RMLn bits
CANx_RML1 CANx_RML2	"Receive Message Lost (CANx_RMLx) Registers" on page 31-72	Write accesses have no effect
CANx_OPSS1 CANx_OPSS2	"Overwrite Protection/Single Shot Transmission (CANx_OPSSx) Register" on page 31-73	Function depends on mailbox direction. Has no effect when RFHn = 1. Do not modify OPSSn bit if mailbox n is enabled
CANx_TRS1 CANx_TRS2	"Transmission Request Set (CANx_TRSx) Registers" on page 31-74	May by set by internal logic under certain circumstances. TRS[7:0] are read-only
CANx_TRR1 CANx_TRR2	"Transmission Request Reset (CANx_TRRx) Registers" on page 31-75	TRRn bits must not be set if mailbox n is disabled or TRSn = 0
CANx_AA1 CANx_AA2	"Abort Acknowledge (CANx_AAx) Registers" on page 31-76	AAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic

Table 31-7. CAN Mailbox Control Registers (Cont'd)

Register Name	Description	Notes
CANx_TA1 CANx_TA2	"Transmission Acknowledge (CANx_TAx) Registers" on page 31-77	TAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CANx_MBTD	"Temporary Mailbox Disable (CANx_MBTD) Register" on page 31-78	Allows safe access to data field of an enabled mailbox
CANx_RFH1 CANx_RFH2	"Remote Frame Handling (CANx_RFHx) Registers" on page 31-78	Available only to configurable mailboxes 23:8. RFH[31:24] and RFH[7:0] are read-only
CANx_MBIM1 CANx_MBIM2	"Mailbox Interrupt Mask (CANx_MBIMx) Registers" on page 31-79	Mailbox interrupts are raised only if these bits are set
CANx_MBTIF 1 CANx_MBTIF 2	"Mailbox Transmit Interrupt Flag (CANx_MBTIFx) Registers" on page 31-80	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBTIFn = 1 results in MBRIFn = 1 and MBTIFn = 0
CANx_MBRIF 1 CANx_MBRIF 2	"Mailbox Receive Interrupt Flag (CANx_MBRIFx) Registers" on page 31-81	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBRIFn = 1 results in MBTIFn = 1 and MBRIFn = 0

Table 31-8. CAN Universal Counter Registers

Register Name	Description	Notes
CANx_UCCNF	"Universal Counter Configuration Mode (CANx_UCCNF) Register" on page 31-83	Bits 15:8 and bit 4 are reserved
CANx_UCCNT	"Universal Counter (CANx_UCCNT) Register" on page 31-84	Counts up or down based on universal counter mode
CANx_UCRC	"Universal Counter Reload/Capture (CANx_UCRC) Register" on page 31-84	In timestamp mode, holds time of last successful transmit or receive

Table 31-9. CAN Error Registers

Register Name	Description	Notes
CANx_CEC	"Error Counter (CANx_CEC) Register" on page 31-84	Undefined while in bus off mode, not affected by soft reset
CANx_ESR	"Error Status (CANx_ESR) Register" on page 31-85	Only the first error is stored. SA0 flag is cleared by recessive bit on CAN bus
CANx_EWR	"Error Counter Warning Level (CANx_EWR) Register" on page 31-85	Default is 96 for each counter

Global CAN Registers

Figure 31-12 through Figure 31-20 on page 31-50 show the CAN global registers.

Master Control (CANx_CONTROL) Registers

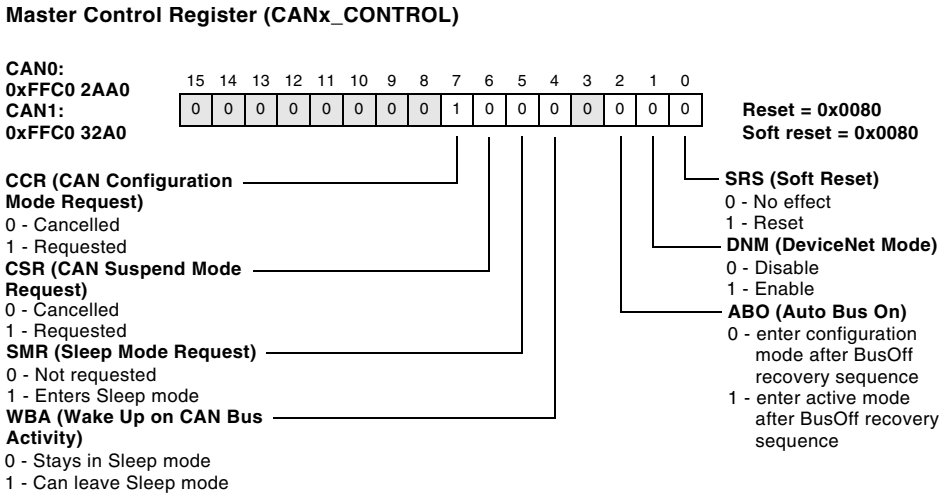


Figure 31-12. Master Control Registers

Global CAN Status (CANx_STATUS) Registers

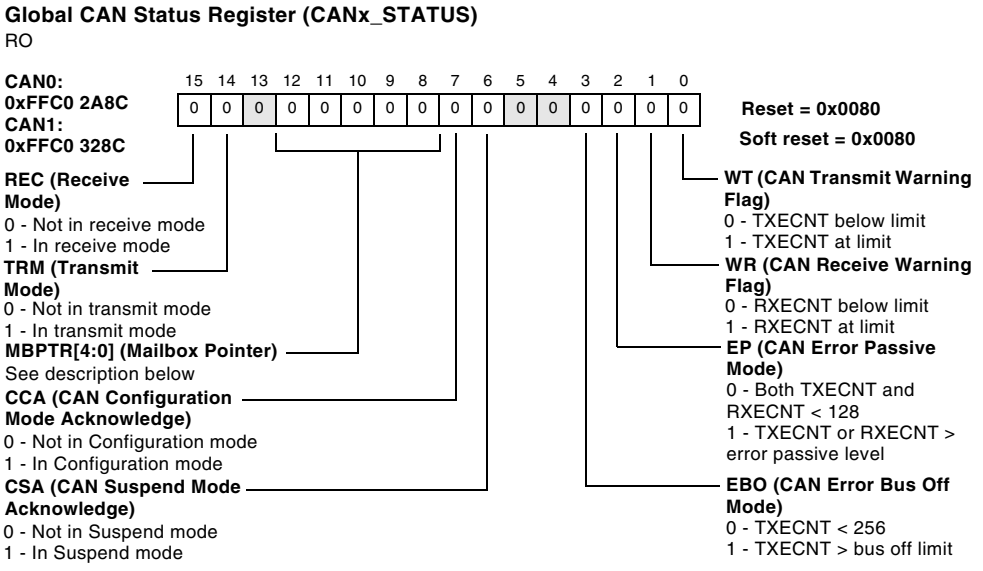


Figure 31-13. Global CAN Status Registers

- **Mail box pointer** (`MBPTR[4:0]`)

  Represents the mailbox number of the current transmit message. After a successful transmission, these bits remain unchanged.

  [b#11111] The message of mailbox 31 is currently being processed.

  …

  [b#00000] The message of mailbox 0 is currently being processed.

CAN Debug (CANx_DEBUG) Registers

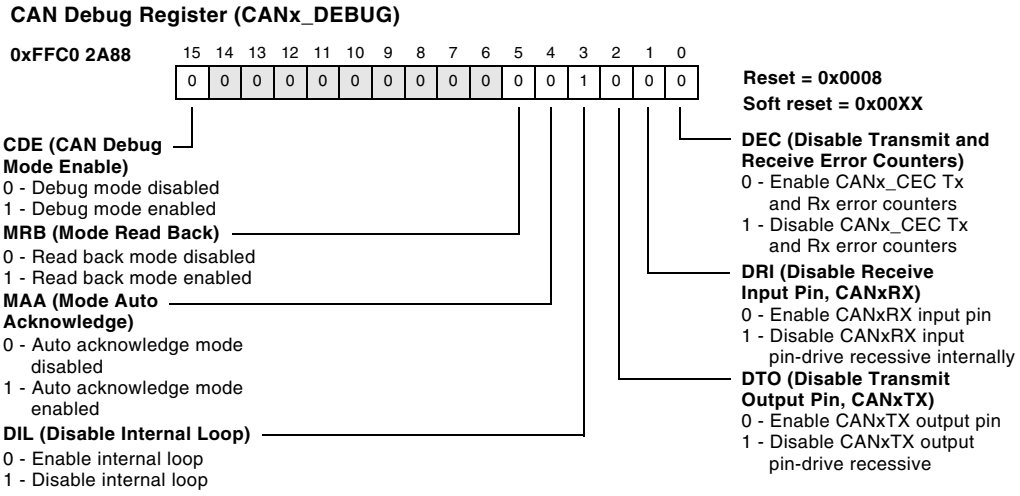


Figure 31-14. CAN Debug Registers

CAN Clock (CANx_CLOCK) Registers

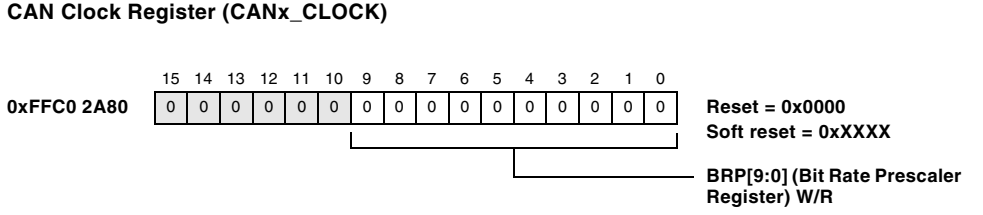


Figure 31-15. CAN Clock Registers

CAN Timing (CANx_TIMING) Registers

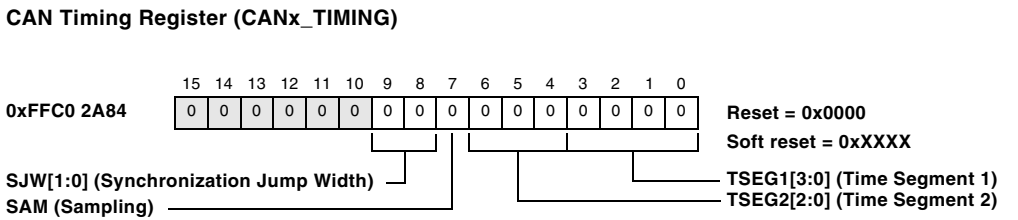


Figure 31-16. CAN Timing Registers

CAN Interrupt (CANx_INTR) Registers

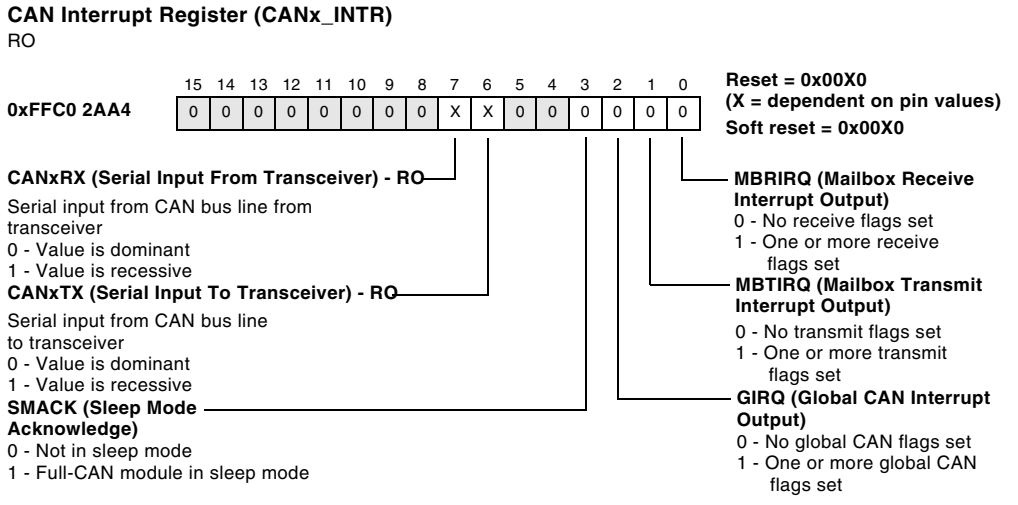


Figure 31-17. CAN Interrupt Registers

Global CAN Interrupt Mask (CANx_GIM) Registers

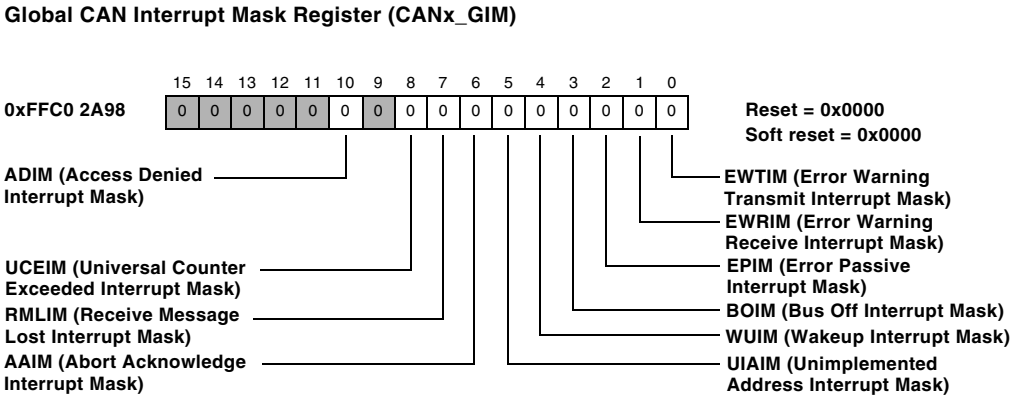


Figure 31-18. Global CAN Interrupt Mask Registers

Global CAN Interrupt Status (CANx_GIS) Registers

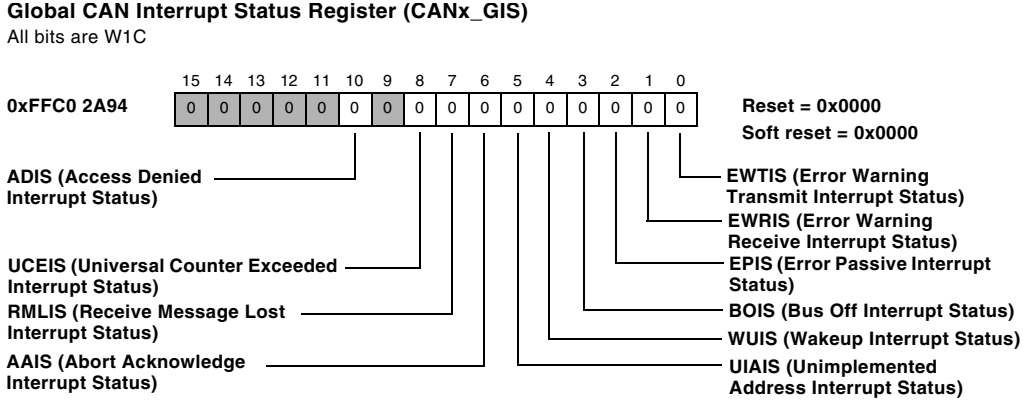


Figure 31-19. Global CAN Interrupt Status Registers

Global CAN Interrupt Flag (CANx_GIF) Registers

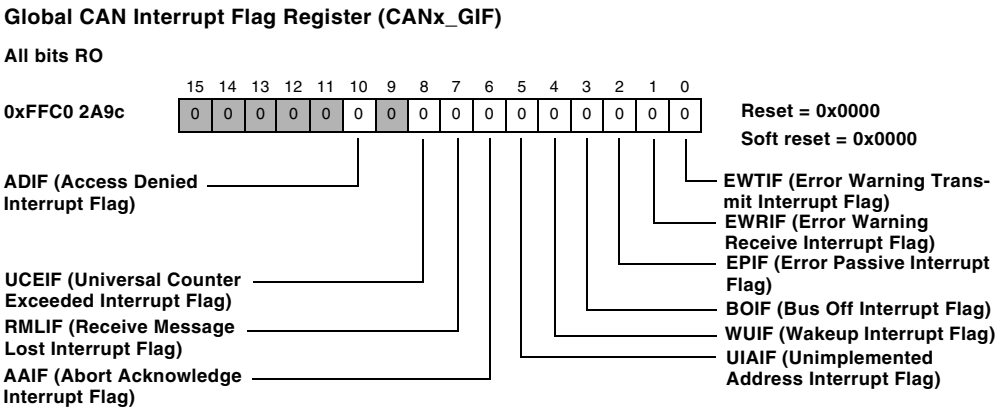


Figure 31-20. Global CAN Interrupt Flag Registers

Mailbox/Mask Registers

Figure 31-21 through Figure 31-30 on page 31-66 show the CAN mailbox and mask registers.

Acceptance Mask (CANx_AMxx) Registers

The value of the acceptance mask register does not matter when the AME bit is zero. If AME is set, only those bits that have the corresponding mask bit cleared are compared to the received message ID. A bit position that is one in the mask register does not need to match.

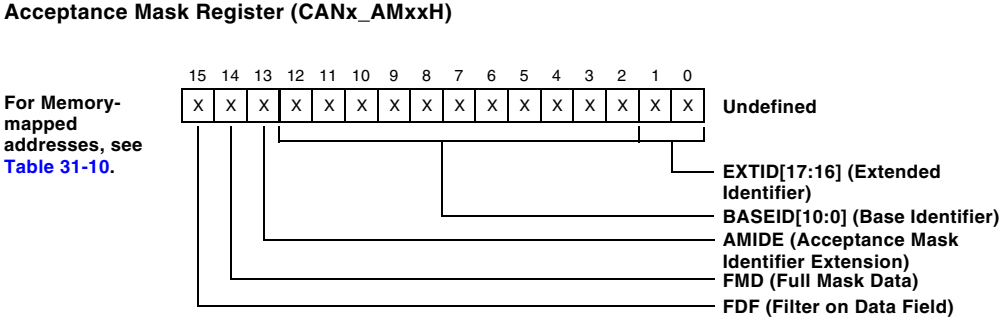


Figure 31-21. Acceptance Mask Registers (H)

Table 31-10. Acceptance Mask Registers (H) Memory-Mapped Addresses

Register Name	CAN0 Memory-mapped Address	CAN1 Memory-mapped Address
CANx_AM00H	0xFFC0 2B04	0xFFC0 3304
CANx_AM01H	0xFFC0 2B0C	0xFFC0 330C
CANx_AM02H	0xFFC0 2B14	0xFFC0 3314
CANx_AM03H	0xFFC0 2B1C	0xFFC0 331C
CANx_AM04H	0xFFC0 2B24	0xFFC0 3324
CANx_AM05H	0xFFC0 2B2C	0xFFC0 332C
CANx_AM06H	0xFFC0 2B34	0xFFC0 3334
CANx_AM07H	0xFFC0 2B3C	0xFFC0 333C
CANx_AM08H	0xFFC0 2B44	0xFFC0 3344
CANx_AM09H	0xFFC0 2B4C	0xFFC0 334C
CANx_AM10H	0xFFC0 2B54	0xFFC0 3354
CANx_AM11H	0xFFC0 2B5C	0xFFC0 335C
CANx_AM12H	0xFFC0 2B64	0xFFC0 3364
CANx_AM13H	0xFFC0 2B6C	0xFFC0 336C
CANx_AM14H	0xFFC0 2B74	0xFFC0 3374

Table 31-10. Acceptance Mask Registers (H) Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address	CAN1 Memory-mapped Address
CANx_AM15H	0xFFC0 2B7C	0xFFC0 337C
CANx_AM16H	0xFFC0 2B84	0xFFC0 3384
CANx_AM17H	0xFFC0 2B8C	0xFFC0 338C
CANx_AM18H	0xFFC0 2B94	0xFFC0 3394
CANx_AM19H	0xFFC0 2B9C	0xFFC0 339C
CANx_AM20H	0xFFC0 2BA4	0xFFC0 33A4
CANx_AM21H	0xFFC0 2BAC	0xFFC0 33AC
CANx_AM22H	0xFFC0 2BB4	0xFFC0 33B4
CANx_AM23H	0xFFC0 2BBC	0xFFC0 33BC
CANx_AM24H	0xFFC0 2BC4	0xFFC0 33C4
CANx_AM25H	0xFFC0 2BCC	0xFFC0 33CC
CANx_AM26H	0xFFC0 2BD4	0xFFC0 33D4
CANx_AM27H	0xFFC0 2BDC	0xFFC0 33DC
CANx_AM28H	0xFFC0 2BE4	0xFFC0 33E4
CANx_AM29H	0xFFC0 2BEC	0xFFC0 33EC
CANx_AM30H	0xFFC0 2BF4	0xFFC0 33F4
CANx_AM31H	0xFFC0 2BFC	0xFFC0 33FC

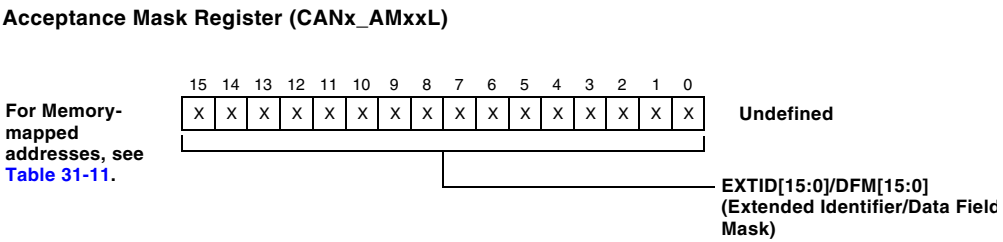


Figure 31-22. Acceptance Mask Registers (L)

Table 31-11. Acceptance Mask Registers (L) Memory-Mapped Addresses

Register Name	CAN0 Memory-mapped Address
CANx_AM00L	0xFFC0 2B00
CANx_AM01L	0xFFC0 2B08
CANx_AM02L	0xFFC0 2B10
CANx_AM03L	0xFFC0 2B18
CANx_AM04L	0xFFC0 2B20
CANx_AM05L	0xFFC0 2B28
CANx_AM06L	0xFFC0 2B30
CANx_AM07L	0xFFC0 2B38
CANx_AM08L	0xFFC0 2B40
CANx_AM09L	0xFFC0 2B48
CANx_AM10L	0xFFC0 2B50
CANx_AM11L	0xFFC0 2B58
CANx_AM12L	0xFFC0 2B60
CANx_AM13L	0xFFC0 2B68
CANx_AM14L	0xFFC0 2B70
CANx_AM15L	0xFFC0 2B78
CANx_AM16L	0xFFC0 2B80
CANx_AM17L	0xFFC0 2B88
CANx_AM18L	0xFFC0 2B90
CANx_AM19L	0xFFC0 2B98
CANx_AM20L	0xFFC0 2BA0
CANx_AM21L	0xFFC0 2BA8
CANx_AM22L	0xFFC0 2BB0
CANx_AM23L	0xFFC0 2BB8
CANx_AM24L	0xFFC0 2BC0

Table 31-11. Acceptance Mask Registers (L) Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address
CANx_AM25L	0xFFC0 2BC8
CANx_AM26L	0xFFC0 2BD0
CANx_AM27L	0xFFC0 2BD8
CANx_AM28L	0xFFC0 2BE0
CANx_AM29L	0xFFC0 2BE8
CANx_AM30L	0xFFC0 2BF0
CANx_AM31L	0xFFC0 2BF8

Mailbox Word 7 (CANx_MBxx_ID1) Registers

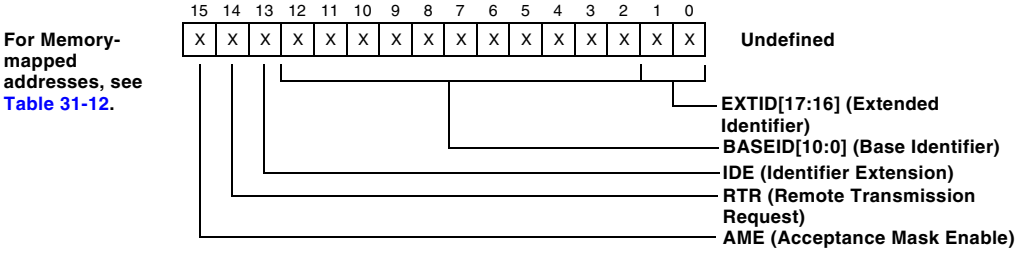


Figure 31-23. Mailbox Word 7 Register

Table 31-12. Mailbox Word 7 Register Memory-Mapped Addresses

Register Name	CAN0 Memory-mapped Address
CANx_MB00_ID1	0xFFC0 2C1C
CANx_MB01_ID1	0xFFC0 2C3C
CANx_MB02_ID1	0xFFC0 2C5C
CANx_MB03_ID1	0xFFC0 2C7C

Table 31-12. Mailbox Word 7 Register Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address
CANx_MB04_ID1	0xFFC0 2C9C
CANx_MB05_ID1	0xFFC0 2CBC
CANx_MB06_ID1	0xFFC0 2CDC
CANx_MB07_ID1	0xFFC0 2CFC
CANx_MB08_ID1	0xFFC0 2D1C
CANx_MB09_ID1	0xFFC0 2D3C
CANx_MB10_ID1	0xFFC0 2D5C
CANx_MB11_ID1	0xFFC0 2D7C
CANx_MB12_ID1	0xFFC0 2D9C
CANx_MB13_ID1	0xFFC0 2DBC
CANx_MB14_ID1	0xFFC0 2DDC
CANx_MB15_ID1	0xFFC0 2DFC
CANx_MB16_ID1	0xFFC0 2E1C
CANx_MB17_ID1	0xFFC0 2E3C
CANx_MB18_ID1	0xFFC0 2E5C
CANx_MB19_ID1	0xFFC0 2E7C
CANx_MB20_ID1	0xFFC0 2E9C
CANx_MB21_ID1	0xFFC0 2EBC
CANx_MB22_ID1	0xFFC0 2EDC
CANx_MB23_ID1	0xFFC0 2EFC
CANx_MB24_ID1	0xFFC0 2F1C
CANx_MB25_ID1	0xFFC0 2F3C
CANx_MB26_ID1	0xFFC0 2F5C
CANx_MB27_ID1	0xFFC0 2F7C
CANx_MB28_ID1	0xFFC0 2F9C

Table 31-12. Mailbox Word 7 Register Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address
CANx_MB29_ID1	0xFFC0 2FBC
CANx_MB30_ID1	0xFFC0 2FDC
CANx_MB31_ID1	0xFFC0 2FFC

Mailbox Word 6 (CANx_MBxx_ID0) Registers

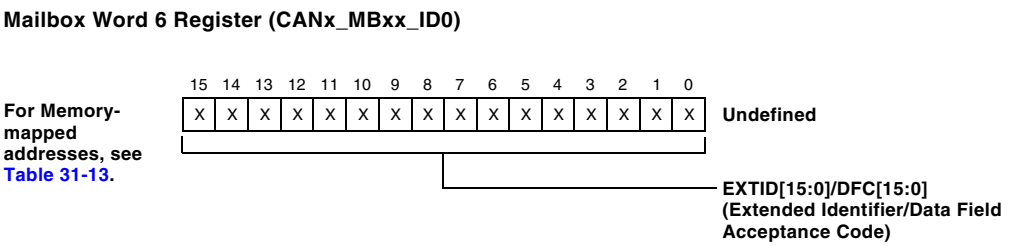


Figure 31-24. Mailbox Word 6 Register

Table 31-13. Mailbox Word 6 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_ID0	0xFFC0 2C18
CANx_MB01_ID0	0xFFC0 2C38
CANx_MB02_ID0	0xFFC0 2C58
CANx_MB03_ID0	0xFFC0 2C78
CANx_MB04_ID0	0xFFC0 2C98
CANx_MB05_ID0	0xFFC0 2CB8
CANx_MB06_ID0	0xFFC0 2CD8
CANx_MB07_ID0	0xFFC0 2CF8
CANx_MB08_ID0	0xFFC0 2D18

Table 31-13. Mailbox Word 6 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB09_ID0	0xFFC0 2D38
CANx_MB10_ID0	0xFFC0 2D58
CANx_MB11_ID0	0xFFC0 2D78
CANx_MB12_ID0	0xFFC0 2D98
CANx_MB13_ID0	0xFFC0 2DB8
CANx_MB14_ID0	0xFFC0 2DD8
CANx_MB15_ID0	0xFFC0 2DF8
CANx_MB16_ID0	0xFFC0 2E18
CANx_MB17_ID0	0xFFC0 2E38
CANx_MB18_ID0	0xFFC0 2E58
CANx_MB19_ID0	0xFFC0 2E78
CANx_MB20_ID0	0xFFC0 2E98
CANx_MB21_ID0	0xFFC0 2EB8
CANx_MB22_ID0	0xFFC0 2ED8
CANx_MB23_ID0	0xFFC0 2EF8
CANx_MB24_ID0	0xFFC0 2F18
CANx_MB25_ID0	0xFFC0 2F38
CANx_MB26_ID0	0xFFC0 2F58
CANx_MB27_ID0	0xFFC0 2F78
CANx_MB28_ID0	0xFFC0 2F98
CANx_MB29_ID0	0xFFC0 2FB8
CANx_MB30_ID0	0xFFC0 2FD8
CANx_MB31_ID0	0xFFC0 2FF8

Mailbox Word 5 (CANx_MBxx_TIMESTAMP) Registers

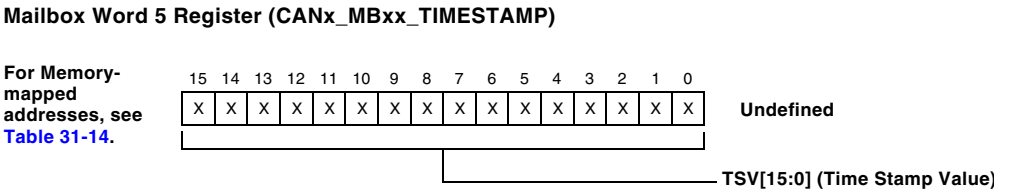


Figure 31-25. Mailbox Word 5 Register

Table 31-14. Mailbox Word 5 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_TIMESTAMP	0xFFC0 2C14
CANx_MB01_TIMESTAMP	0xFFC0 2C34
CANx_MB02_TIMESTAMP	0xFFC0 2C54
CANx_MB03_TIMESTAMP	0xFFC0 2C74
CANx_MB04_TIMESTAMP	0xFFC0 2C94
CANx_MB05_TIMESTAMP	0xFFC0 2CB4
CANx_MB06_TIMESTAMP	0xFFC0 2CD4
CANx_MB07_TIMESTAMP	0xFFC0 2CF4
CANx_MB08_TIMESTAMP	0xFFC0 2D14
CANx_MB09_TIMESTAMP	0xFFC0 2D34
CANx_MB10_TIMESTAMP	0xFFC0 2D54
CANx_MB11_TIMESTAMP	0xFFC0 2D74
CANx_MB12_TIMESTAMP	0xFFC0 2D94
CANx_MB13_TIMESTAMP	0xFFC0 2DB4
CANx_MB14_TIMESTAMP	0xFFC0 2DD4
CANx_MB15_TIMESTAMP	0xFFC0 2DF4

Table 31-14. Mailbox Word 5 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB16_TIMESTAMP	0xFFC0 2E14
CANx_MB17_TIMESTAMP	0xFFC0 2E34
CANx_MB18_TIMESTAMP	0xFFC0 2E54
CANx_MB19_TIMESTAMP	0xFFC0 2E74
CANx_MB20_TIMESTAMP	0xFFC0 2E94
CANx_MB21_TIMESTAMP	0xFFC0 2EB4
CANx_MB22_TIMESTAMP	0xFFC0 2ED4
CANx_MB23_TIMESTAMP	0xFFC0 2EF4
CANx_MB24_TIMESTAMP	0xFFC0 2F14
CANx_MB25_TIMESTAMP	0xFFC0 2F34
CANx_MB26_TIMESTAMP	0xFFC0 2F54
CANx_MB27_TIMESTAMP	0xFFC0 2F74
CANx_MB28_TIMESTAMP	0xFFC0 2F94
CANx_MB29_TIMESTAMP	0xFFC0 2FB4
CANx_MB30_TIMESTAMP	0xFFC0 2FD4
CANx_MB31_TIMESTAMP	0xFFC0 2FF4

Mailbox Word 4 (CANx_MBxx_LENGTH) Registers

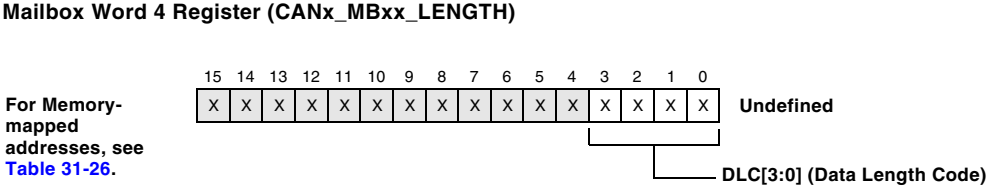


Figure 31-26. Mailbox Word 4 Register

Table 31-15. Mailbox Word 4 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_LENGTH	0xFFC0 2C10
CANx_MB01_LENGTH	0xFFC0 2C30
CANx_MB02_LENGTH	0xFFC0 2C50
CANx_MB03_LENGTH	0xFFC0 2C70
CANx_MB04_LENGTH	0xFFC0 2C90
CANx_MB05_LENGTH	0xFFC0 2CB0
CANx_MB06_LENGTH	0xFFC0 2CD0
CANx_MB07_LENGTH	0xFFC0 2CF0
CANx_MB08_LENGTH	0xFFC0 2D10
CANx_MB09_LENGTH	0xFFC0 2D30
CANx_MB10_LENGTH	0xFFC0 2D50
CANx_MB11_LENGTH	0xFFC0 2D70
CANx_MB12_LENGTH	0xFFC0 2D90
CANx_MB13_LENGTH	0xFFC0 2DB0
CANx_MB14_LENGTH	0xFFC0 2DD0
CANx_MB15_LENGTH	0xFFC0 2DF0
CANx_MB16_LENGTH	0xFFC0 2E10
CANx_MB17_LENGTH	0xFFC0 2E30
CANx_MB18_LENGTH	0xFFC0 2E50
CANx_MB19_LENGTH	0xFFC0 2E70
CANx_MB20_LENGTH	0xFFC0 2E90
CANx_MB21_LENGTH	0xFFC0 2EB0
CANx_MB22_LENGTH	0xFFC0 2ED0
CANx_MB23_LENGTH	0xFFC0 2EF0
CANx_MB24_LENGTH	0xFFC0 2F10

Table 31-15. Mailbox Word 4 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB25_LENGTH	0xFFC0 2F30
CANx_MB26_LENGTH	0xFFC0 2F50
CANx_MB27_LENGTH	0xFFC0 2F70
CANx_MB28_LENGTH	0xFFC0 2F90
CANx_MB29_LENGTH	0xFFC0 2FB0
CANx_MB30_LENGTH	0xFFC0 2FD0
CANx_MB31_LENGTH	0xFFC0 2FF0

Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers

The following are the descriptions of Mailbox Word registers (CANx_MBxx_DATA3/2/1/0) and their appropriate memory-mapped addresses.

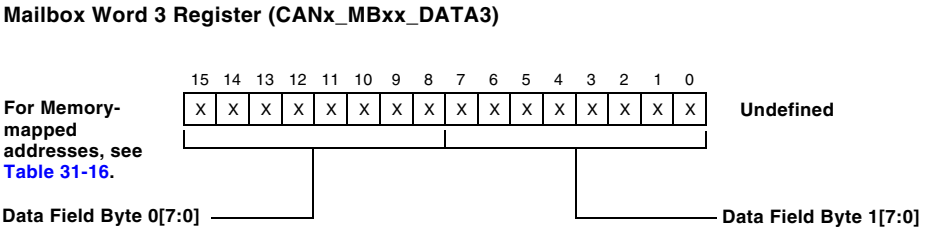


Figure 31-27. Mailbox Word 3 Register

Table 31-16. Mailbox Word 3 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_DATA3	0xFFC0 2C0C
CANx_MB01_DATA3	0xFFC0 2C2C

Table 31-16. Mailbox Word 3 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB02_DATA3	0xFFC0 2C4C
CANx_MB03_DATA3	0xFFC0 2C6C
CANx_MB04_DATA3	0xFFC0 2C8C
CANx_MB05_DATA3	0xFFC0 2CAC
CANx_MB06_DATA3	0xFFC0 2CCC
CANx_MB07_DATA3	0xFFC0 2CEC
CANx_MB08_DATA3	0xFFC0 2D0C
CANx_MB09_DATA3	0xFFC0 2D2C
CANx_MB10_DATA3	0xFFC0 2D4C
CANx_MB11_DATA3	0xFFC0 2D6C
CANx_MB12_DATA3	0xFFC0 2D8C
CANx_MB13_DATA3	0xFFC0 2DAC
CANx_MB14_DATA3	0xFFC0 2DCC
CANx_MB15_DATA3	0xFFC0 2DEC
CANx_MB16_DATA3	0xFFC0 2E0C
CANx_MB17_DATA3	0xFFC0 2E2C
CANx_MB18_DATA3	0xFFC0 2E4C
CANx_MB19_DATA3	0xFFC0 2E6C
CANx_MB20_DATA3	0xFFC0 2E8C
CANx_MB21_DATA3	0xFFC0 2EAC
CANx_MB22_DATA3	0xFFC0 2ECC
CANx_MB23_DATA3	0xFFC0 2EEC
CANx_MB24_DATA3	0xFFC0 2F0C
CANx_MB25_DATA3	0xFFC0 2F2C
CANx_MB26_DATA3	0xFFC0 2F4C

Table 31-16. Mailbox Word 3 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB27_DATA3	0xFFC0 2F6C
CANx_MB28_DATA3	0xFFC0 2F8C
CANx_MB29_DATA3	0xFFC0 2FAC
CANx_MB30_DATA3	0xFFC0 2FCC
CANx_MB31_DATA3	0xFFC0 2FEC

Mailbox Word 2 Register (CANx_MBxx_DATA2)

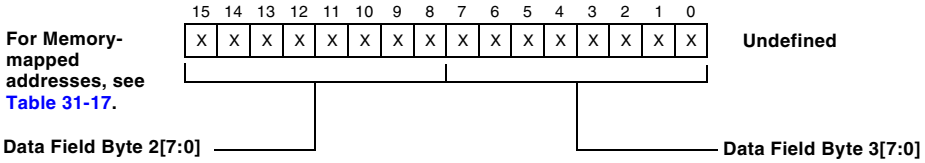


Figure 31-28. Mailbox Word 2 Register

Table 31-17. Mailbox Word 2 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_DATA2	0xFFC0 2C08
CANx_MB01_DATA2	0xFFC0 2C28
CANx_MB02_DATA2	0xFFC0 2C48
CANx_MB03_DATA2	0xFFC0 2C68
CANx_MB04_DATA2	0xFFC0 2C88
CANx_MB05_DATA2	0xFFC0 2CA8
CANx_MB06_DATA2	0xFFC0 2CC8
CANx_MB07_DATA2	0xFFC0 2CE8
CANx_MB08_DATA2	0xFFC0 2D08

Table 31-17. Mailbox Word 2 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB09_DATA2	0xFFC0 2D28
CANx_MB10_DATA2	0xFFC0 2D48
CANx_MB11_DATA2	0xFFC0 2D68
CANx_MB12_DATA2	0xFFC0 2D88
CANx_MB13_DATA2	0xFFC0 2DA8
CANx_MB14_DATA2	0xFFC0 2DC8
CANx_MB15_DATA2	0xFFC0 2DE8
CANx_MB16_DATA2	0xFFC0 2E08
CANx_MB17_DATA2	0xFFC0 2E28
CANx_MB18_DATA2	0xFFC0 2E48
CANx_MB19_DATA2	0xFFC0 2E68
CANx_MB20_DATA2	0xFFC0 2E88
CANx_MB21_DATA2	0xFFC0 2EA8
CANx_MB22_DATA2	0xFFC0 2EC8
CANx_MB23_DATA2	0xFFC0 2EE8
CANx_MB24_DATA2	0xFFC0 2F08
CANx_MB25_DATA2	0xFFC0 2F28
CANx_MB26_DATA2	0xFFC0 2F48
CANx_MB27_DATA2	0xFFC0 2F68
CANx_MB28_DATA2	0xFFC0 2F88
CANx_MB29_DATA2	0xFFC0 2FA8
CANx_MB30_DATA2	0xFFC0 2FC8
CANx_MB31_DATA2	0xFFC0 2FE8

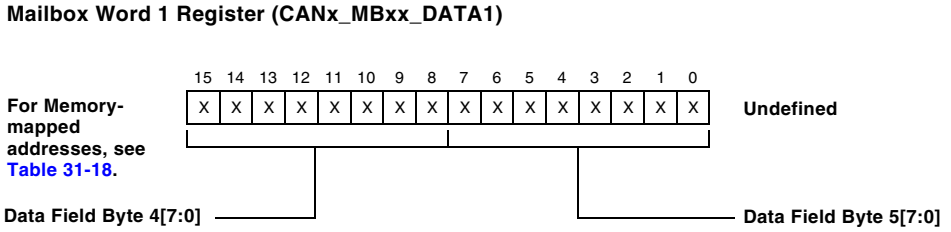


Figure 31-29. Mailbox Word 1 Register

Table 31-18. Mailbox Word 1 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_DATA1	0xFFC0 2C04
CANx_MB01_DATA1	0xFFC0 2C24
CANx_MB02_DATA1	0xFFC0 2C44
CANx_MB03_DATA1	0xFFC0 2C64
CANx_MB04_DATA1	0xFFC0 2C84
CANx_MB05_DATA1	0xFFC0 2CA4
CANx_MB06_DATA1	0xFFC0 2CC4
CANx_MB07_DATA1	0xFFC0 2CE4
CANx_MB08_DATA1	0xFFC0 2D04
CANx_MB09_DATA1	0xFFC0 2D24
CANx_MB10_DATA1	0xFFC0 2D44
CANx_MB11_DATA1	0xFFC0 2D64
CANx_MB12_DATA1	0xFFC0 2D84
CANx_MB13_DATA1	0xFFC0 2DA4
CANx_MB14_DATA1	0xFFC0 2DC4
CANx_MB15_DATA1	0xFFC0 2DE4
CANx_MB16_DATA1	0xFFC0 2E04
CANx_MB17_DATA1	0xFFC0 2E24

Table 31-18. Mailbox Word 1 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB18_DATA1	0xFFC0 2E44
CANx_MB19_DATA1	0xFFC0 2E64
CANx_MB20_DATA1	0xFFC0 2E84
CANx_MB21_DATA1	0xFFC0 2EA4
CANx_MB22_DATA1	0xFFC0 2EC4
CANx_MB23_DATA1	0xFFC0 2EE4
CANx_MB24_DATA1	0xFFC0 2F04
CANx_MB25_DATA1	0xFFC0 2F24
CANx_MB26_DATA1	0xFFC0 2F44
CANx_MB27_DATA1	0xFFC0 2F64
CANx_MB28_DATA1	0xFFC0 2F84
CANx_MB29_DATA1	0xFFC0 2FA4
CANx_MB30_DATA1	0xFFC0 2FC4
CANx_MB31_DATA1	0xFFC0 2FE4

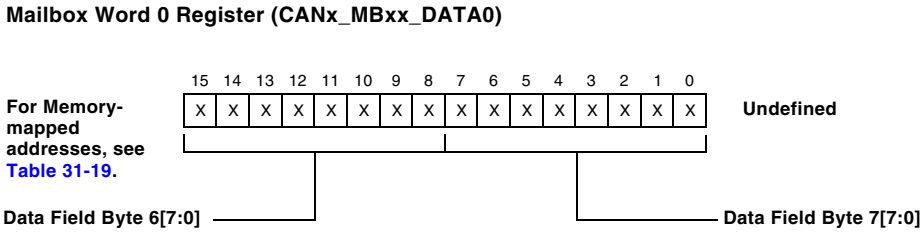


Figure 31-30. Mailbox Word 0 Register

Table 31-19. Mailbox Word 0 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_DATA0	0xFFC0 2C00
CANx_MB01_DATA0	0xFFC0 2C20
CANx_MB02_DATA0	0xFFC0 2C40
CANx_MB03_DATA0	0xFFC0 2C60
CANx_MB04_DATA0	0xFFC0 2C80
CANx_MB05_DATA0	0xFFC0 2CA0
CANx_MB06_DATA0	0xFFC0 2CC0
CANx_MB07_DATA0	0xFFC0 2CE0
CANx_MB08_DATA0	0xFFC0 2D00
CANx_MB09_DATA0	0xFFC0 2D20
CANx_MB10_DATA0	0xFFC0 2D40
CANx_MB11_DATA0	0xFFC0 2D60
CANx_MB12_DATA0	0xFFC0 2D80
CANx_MB13_DATA0	0xFFC0 2DA0
CANx_MB14_DATA0	0xFFC0 2DC0
CANx_MB15_DATA0	0xFFC0 2DE0
CANx_MB16_DATA0	0xFFC0 2E00
CANx_MB17_DATA0	0xFFC0 2E20
CANx_MB18_DATA0	0xFFC0 2E40
CANx_MB19_DATA0	0xFFC0 2E60
CANx_MB20_DATA0	0xFFC0 2E80
CANx_MB21_DATA0	0xFFC0 2EA0
CANx_MB22_DATA0	0xFFC0 2EC0
CANx_MB23_DATA0	0xFFC0 2EE0
CANx_MB24_DATA0	0xFFC0 2F00

Table 31-19. Mailbox Word 0 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB25_DATA0	0xFFC0 2F20
CANx_MB26_DATA0	0xFFC0 2F40
CANx_MB27_DATA0	0xFFC0 2F60
CANx_MB28_DATA0	0xFFC0 2F80
CANx_MB29_DATA0	0xFFC0 2FA0
CANx_MB30_DATA0	0xFFC0 2FC0
CANx_MB31_DATA0	0xFFC0 2FE0

Mailbox Control Registers

Figure 31-31 through Figure 31-57 on page 31-82 show the mailbox control registers.

Mailbox Configuration (CANx_MCx) Registers

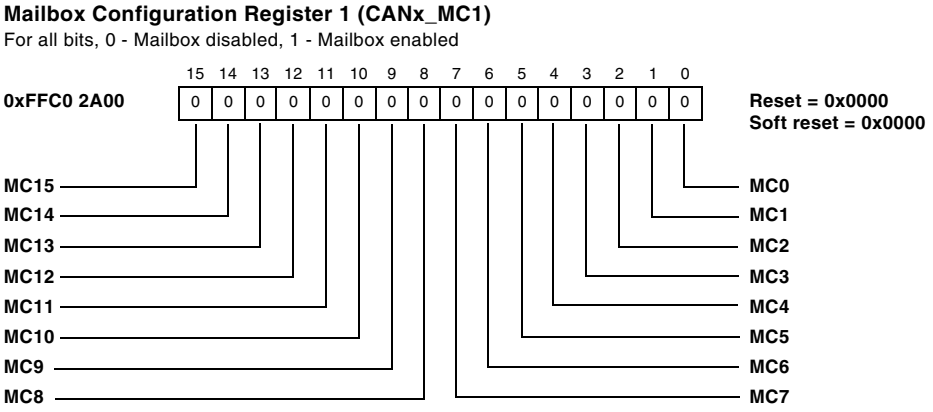


Figure 31-31. Mailbox Configuration Register 1

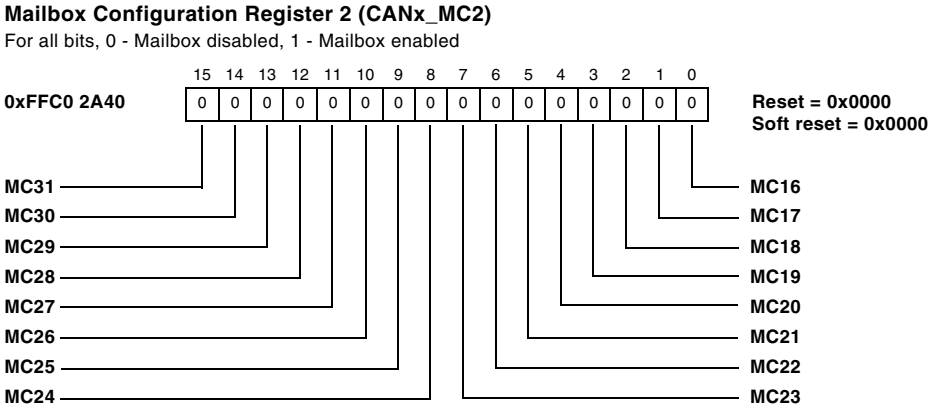


Figure 31-32. Mailbox Configuration Register 2

Mailbox Direction (CANx_MDx) Registers

Mailbox Direction Register 1 (CANx_MD1)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

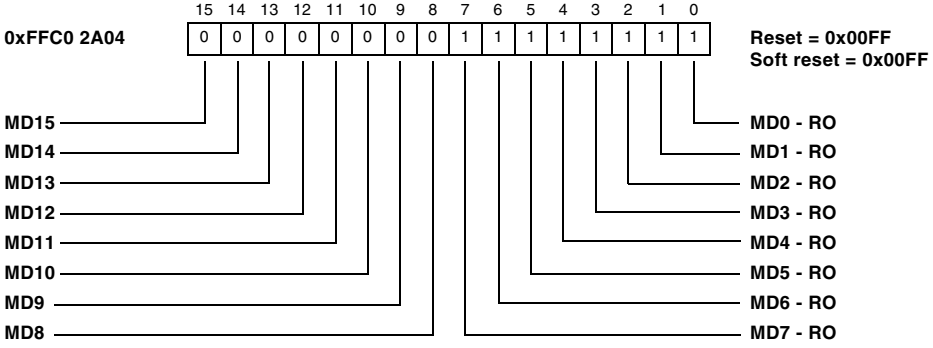


Figure 31-33. Mailbox Direction Register 1

Mailbox Direction Register 2 (CANx_MD2)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

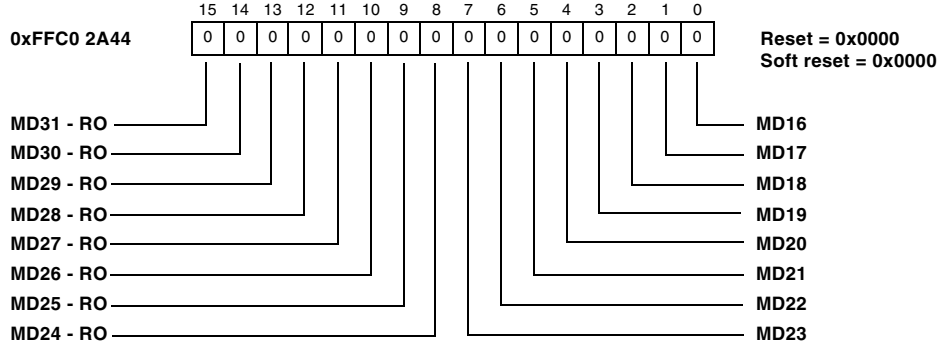


Figure 31-34. Mailbox Direction Register 2

Receive Message Pending (CANx_RMPx) Registers

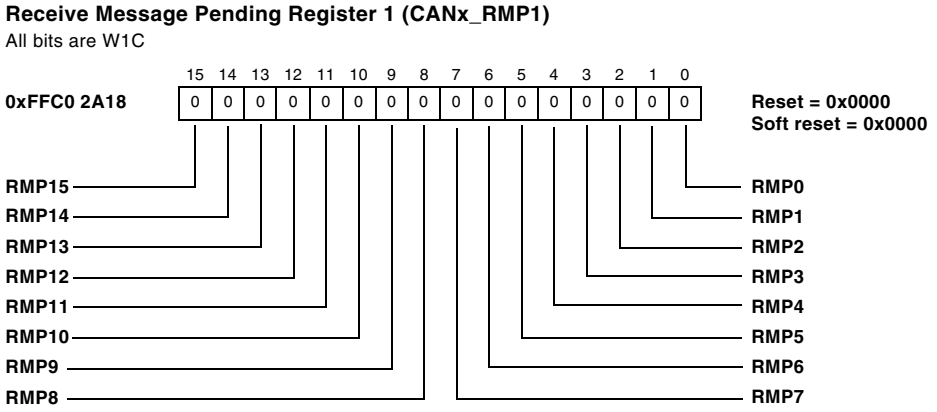


Figure 31-35. Receive Message Pending Register 1

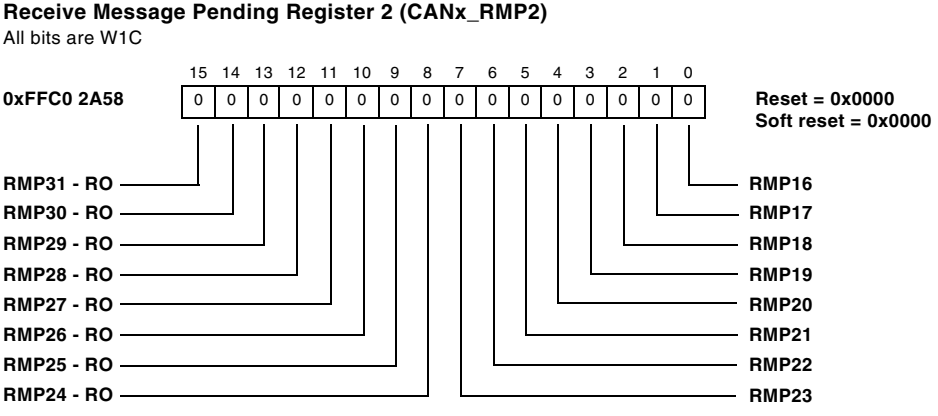


Figure 31-36. Receive Message Pending Register 2

Receive Message Lost (CANx_RMLx) Registers

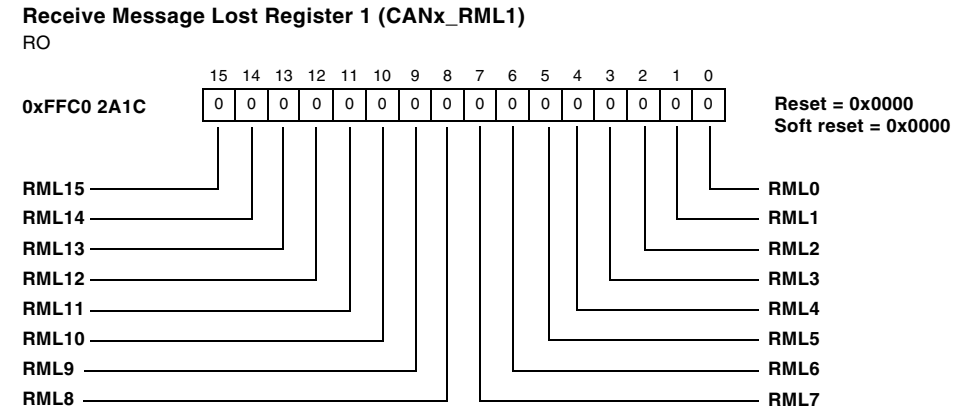


Figure 31-37. Receive Message Lost Register 1

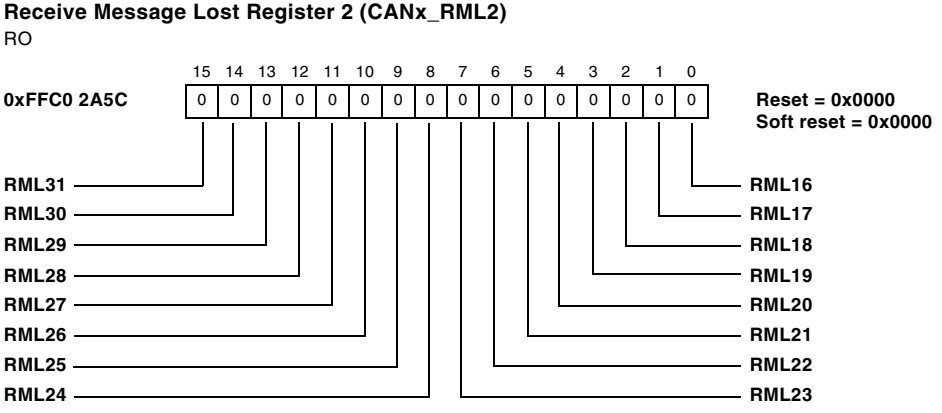


Figure 31-38. Receive Message Lost Register 2

Overwrite Protection/Single Shot Transmission (CANx_OPSSx) Register

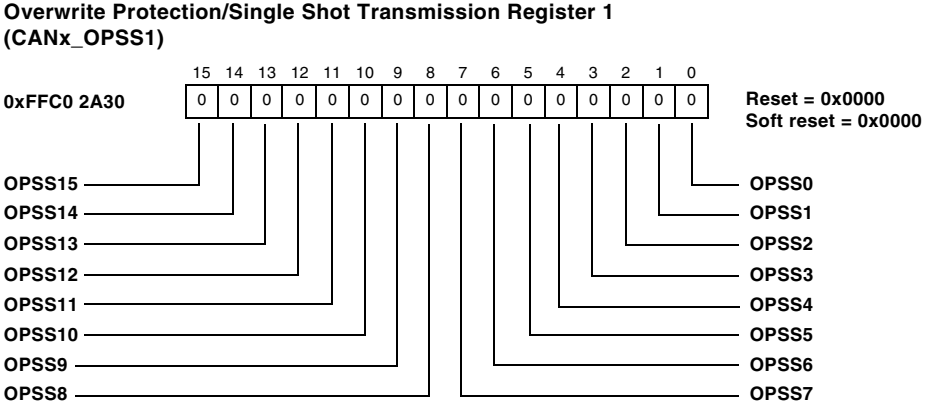


Figure 31-39. Overwrite Protection/Single Shot Transmission Register 1

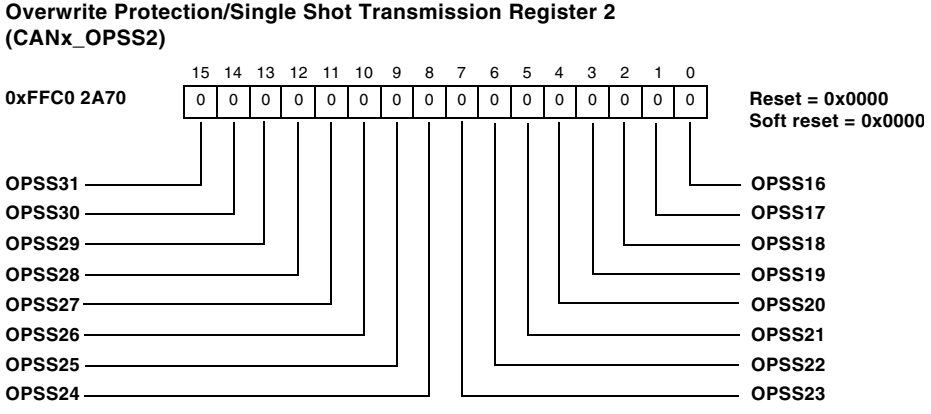


Figure 31-40. Overwrite Protection/Single Shot Transmission Register 2

Transmission Request Set (CANx_TRSx) Registers

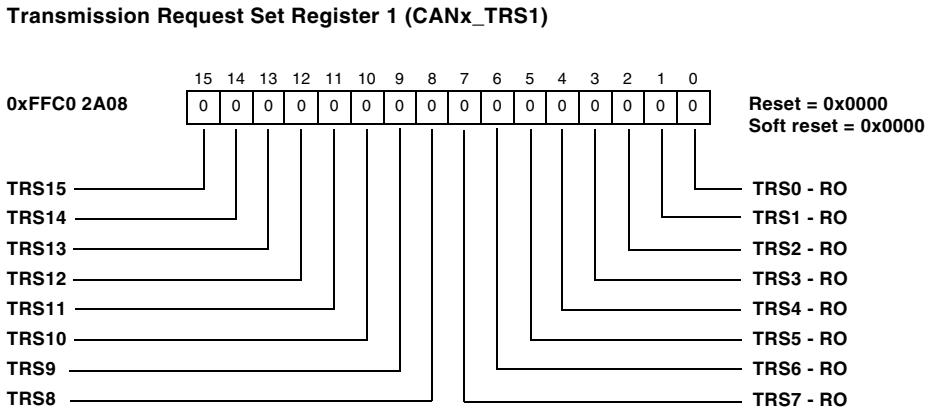


Figure 31-41. Transmission Request Set Register 1

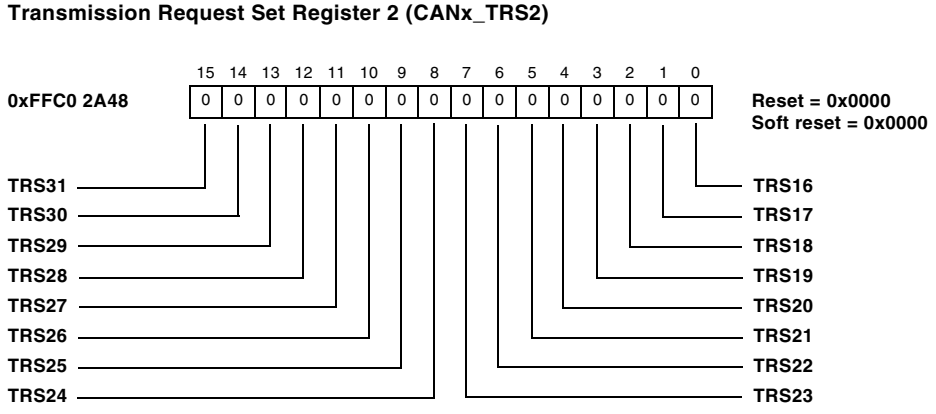


Figure 31-42. Transmission Request Set Register 2

Transmission Request Reset (CANx_TRRx) Registers

Transmission Request Reset Register 1 (CANx_TRR1)

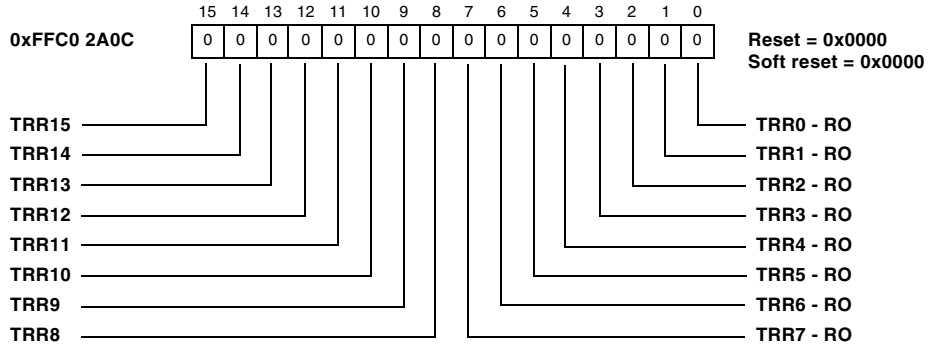


Figure 31-43. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CANx_TRR2)

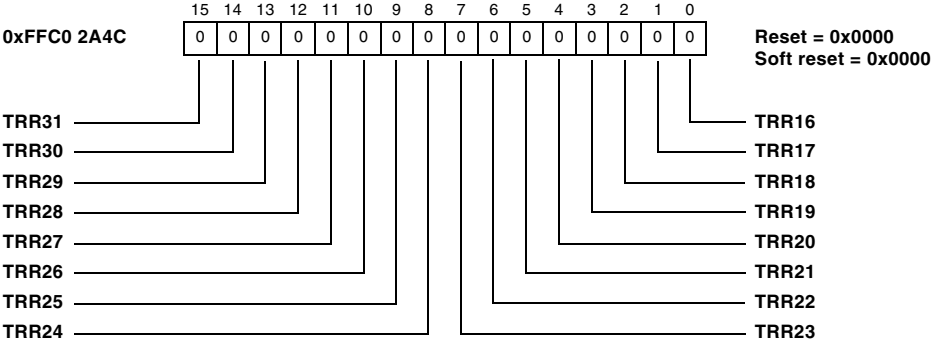


Figure 31-44. Transmission Request Reset Register 2

Abort Acknowledge (CANx_AAx) Registers

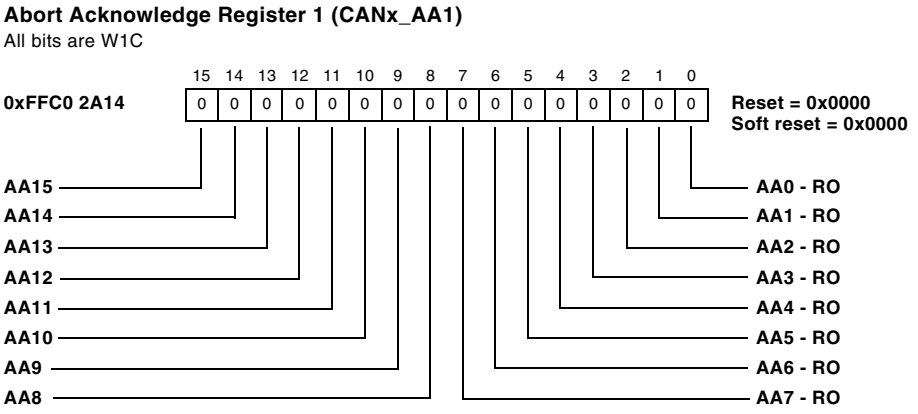


Figure 31-45. Abort Acknowledge Register 1

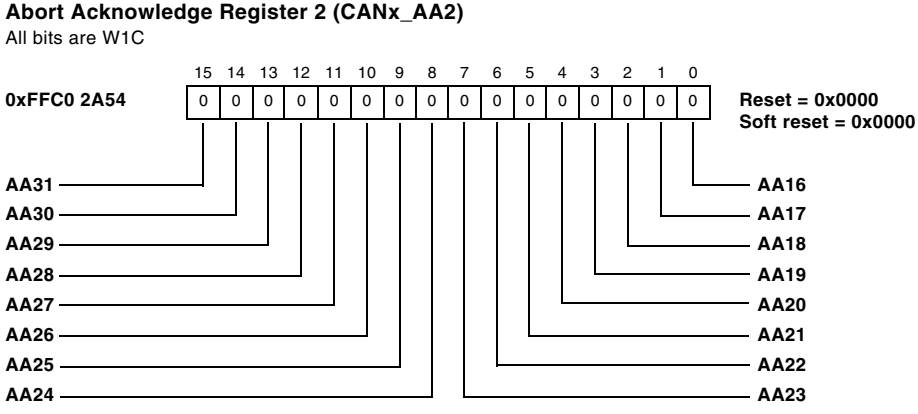


Figure 31-46. Abort Acknowledge Register 2

Transmission Acknowledge (CANx_TAx) Registers

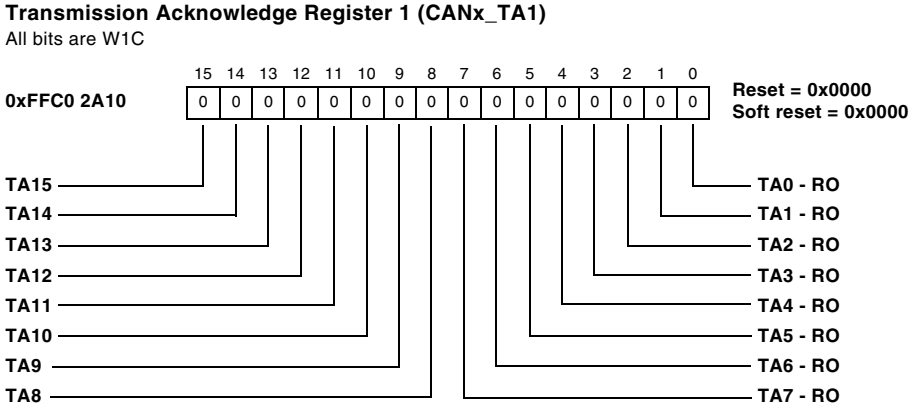


Figure 31-47. Transmission Acknowledge Register 1

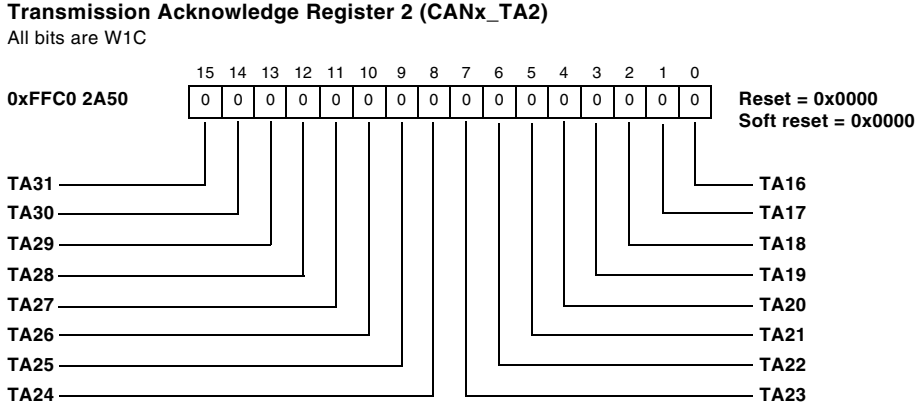


Figure 31-48. Transmission Acknowledge Register 2

Temporary Mailbox Disable (CANx_MBTD) Register

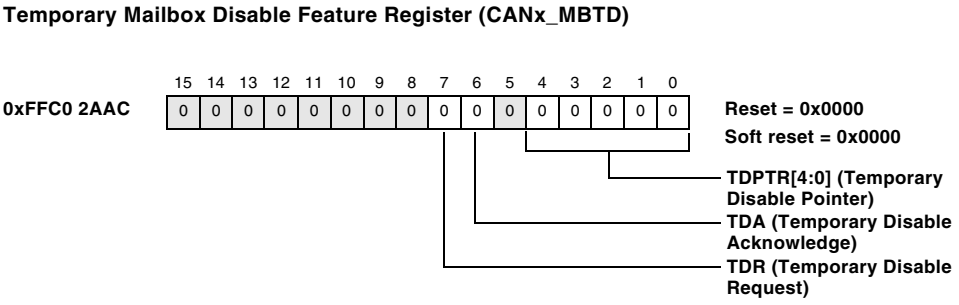


Figure 31-49. Temporary Mailbox Disable Register

Remote Frame Handling (CANx_RFHx) Registers

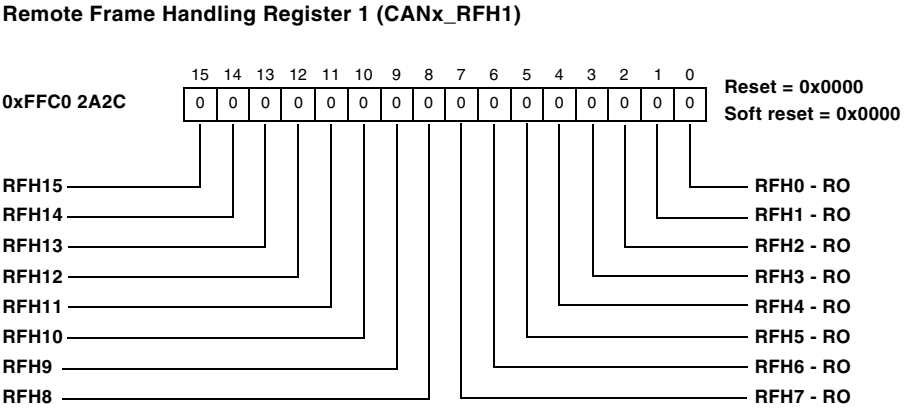


Figure 31-50. Remote Frame Handling Register 1

Remote Frame Handling Register 2 (CANx_RFH2)

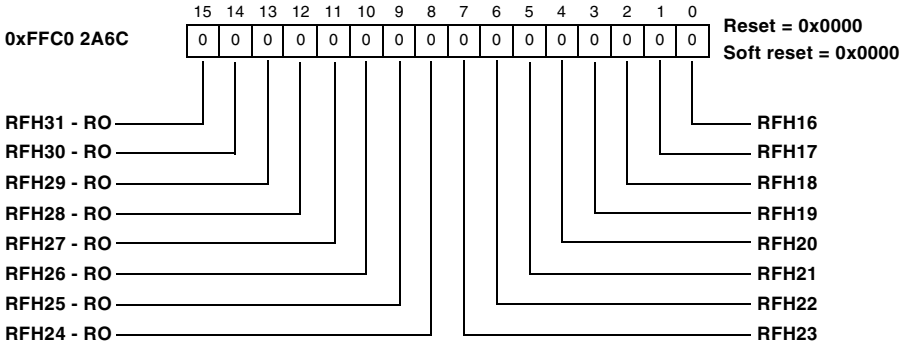


Figure 31-51. Remote Frame Handling Register 2

Mailbox Interrupt Mask (CANx_MBIMx) Registers

Mailbox Interrupt Mask Register 1 (CANx_MBIM1)

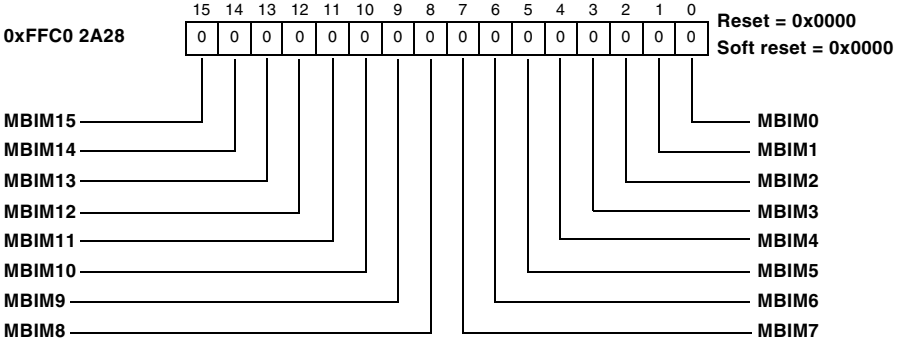


Figure 31-52. Mailbox Interrupt Mask Register 1

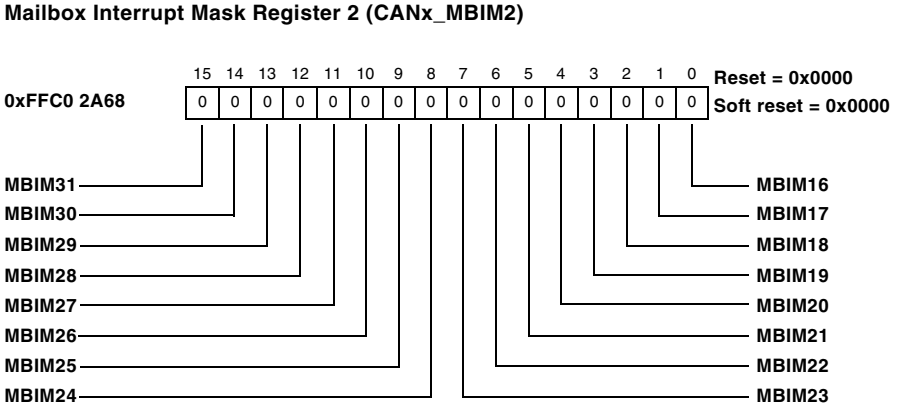


Figure 31-53. Mailbox Interrupt Mask Register 2

Mailbox Transmit Interrupt Flag (CANx_MBTIFx) Registers

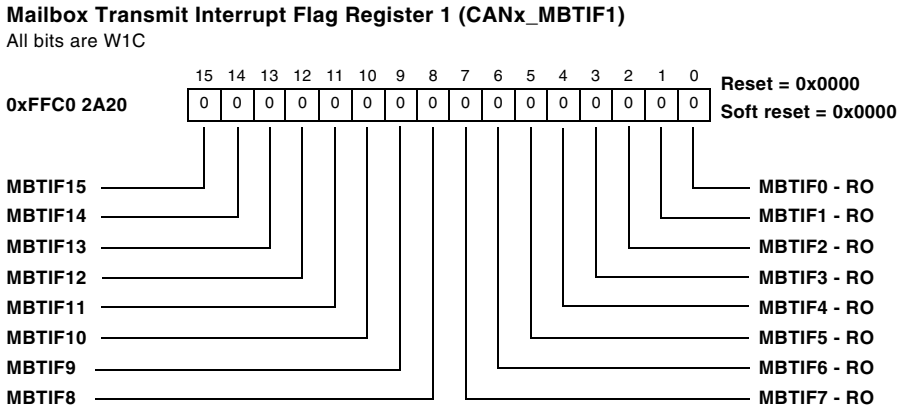


Figure 31-54. Mailbox Transmit Interrupt Flag Register 1

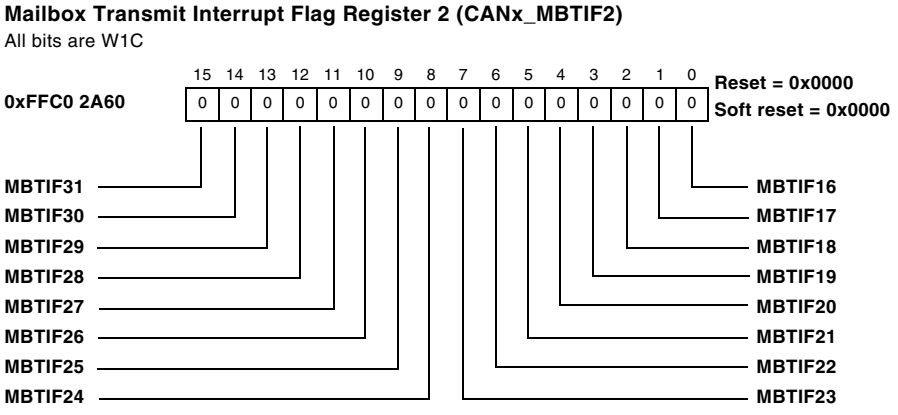


Figure 31-55. Mailbox Transmit Interrupt Flag Register 2

Mailbox Receive Interrupt Flag (CANx_MBRIFx) Registers

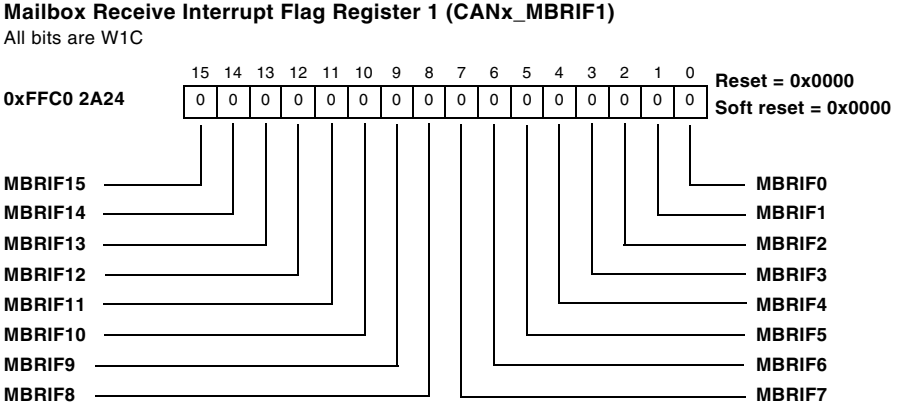


Figure 31-56. Mailbox Receive Interrupt Flag Register 1

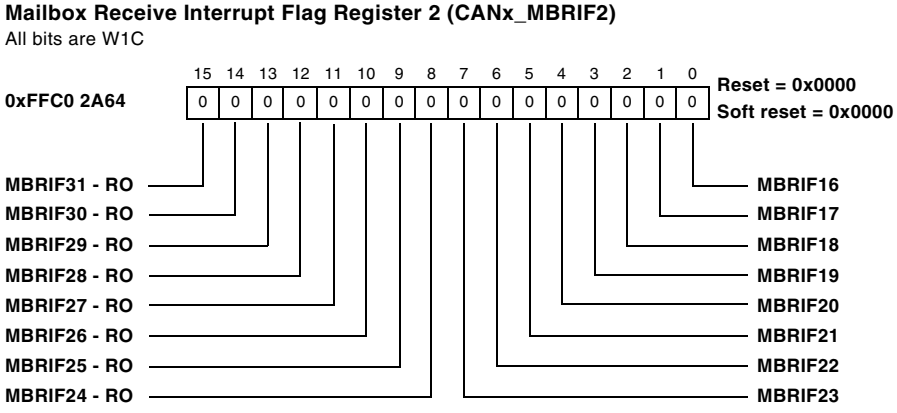


Figure 31-57. Mailbox Receive Interrupt Flag Register 2

Universal Counter Registers

Figure 31-58 through Figure 31-60 show the universal counter registers.

Universal Counter Configuration Mode (CANx_UCCNF) Register

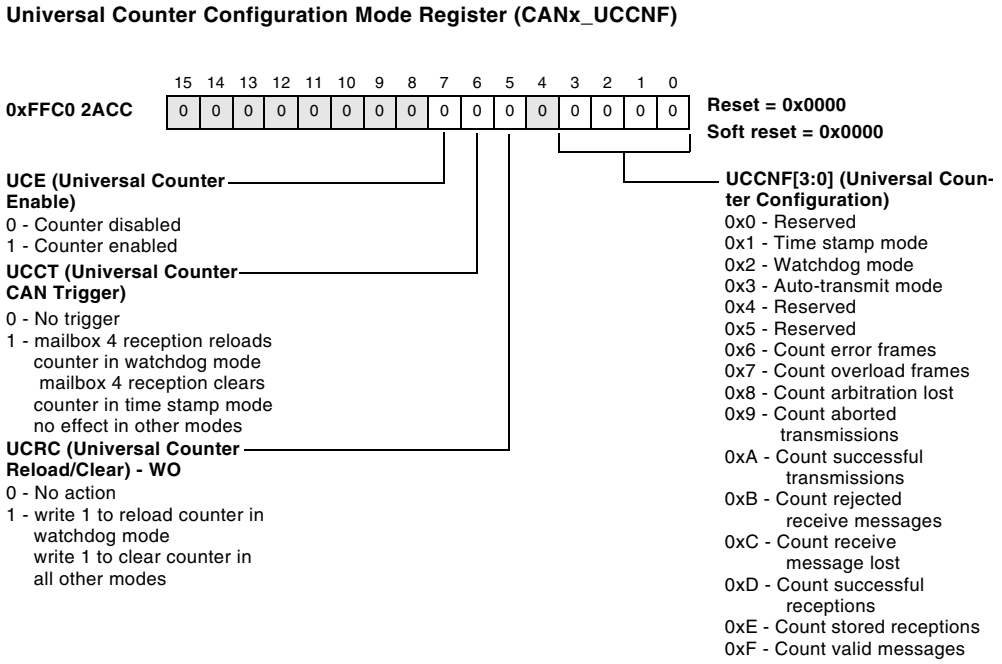


Figure 31-58. Universal Counter Configuration Mode Register

Universal Counter (CANx_UCCNT) Register

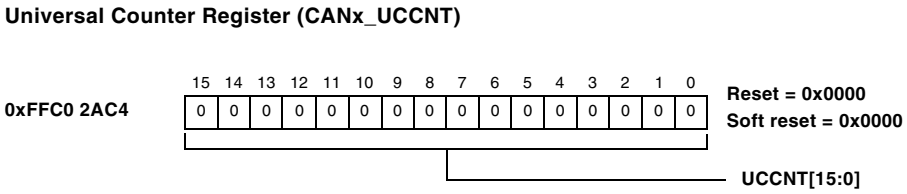


Figure 31-59. Universal Counter Register

Universal Counter Reload/Capture (CANx_UCRC) Register

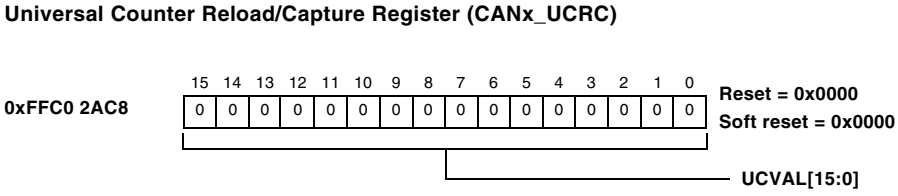


Figure 31-60. Universal Counter Reload/Capture Register

Error Registers

Figure 31-61 through Figure 31-63 show the CAN controller error registers.

Error Counter (CANx_CEC) Register

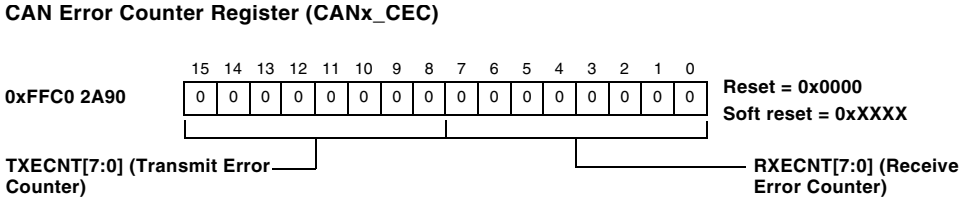


Figure 31-61. Error Counter Register

Error Status (CANx_ESR) Register

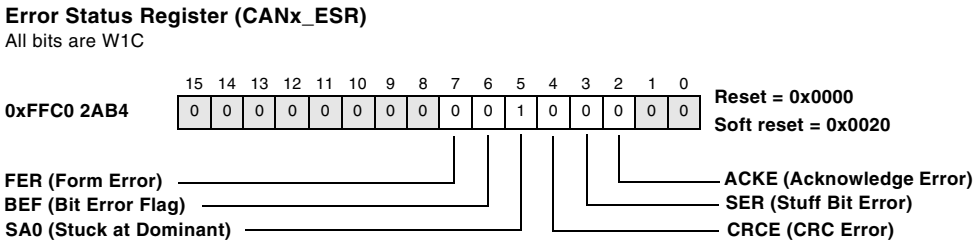


Figure 31-62. Error Status Register

Error Counter Warning Level (CANx_EWR) Register

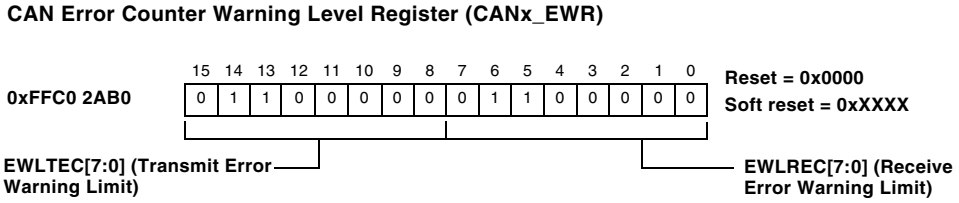


Figure 31-63. Error Counter Warning Level Register

Programming Examples

The following CAN code examples (Listing 31-2 through Listing 31-4 on page 31-89) 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 <defBF549.h>` for ADSP-BF549 projects).

CAN Setup Code

The following code initializes the port pins to connect to the CAN0 controller and configures the CAN timing parameters.

Listing 31-2. Initializing CAN0

```
Initialize_CAN0:
  P0.H = HI(PORTG_FER);  /* CAN pins multiplexed on Port G */
  P0.L = LO(PORTG_FER);
  R0 = 0x3000 (Z);  /* Enable CAN0 TX/RX pins on PG12/PG13 */
  W[P0] = R0;
  SSYNC;

  /* ================================================
  ** Set CAN Bit Timing
  **
  ** CANx_TIMING - SJW, TSEG2, and TSEG1 governed by:
  ** SJW <= TSEG2 <= TSEG1
  **
  ** ================================================
  */
  P0.H = HI(CAN0_TIMING);
  P0.L = LO(CAN0_TIMING);
  R0 = 0x0334(Z);  /* SJW = 3, TSEG2 = 3, TSEG1 = 4 */
  W[P0] = R0;
  SSYNC;
 /* ================================================
  ** CANx_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-BF54x 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(CAN0_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 31-3. Initializing and Enabling Mailboxes

```
CAN0_Initialize_Mailboxes:
  P0.H = HI(CAN0_MD1);  /* Configure Mailbox Direction */
  P0.L = LO(CAN0_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(CAN0_MB_ID1(8));  /* Access MB08 ID1 Register */
  W[P0] = R0;  /* Remote frame disabled, 11 bit ID */

  R0 = 0;
  P0.L = LO(CAN0_MB_ID0(8));
  W[P0] = R0;  /* Zero Out Lower ID Register */

  R0 = 4;
  P0.L = LO(CAN0_MB_LENGTH(8));
  W[P0] = R0;  /* Set DLC to 4 Bytes */
  R0 = 0xAABB(Z);
  P0.L = LO(CAN0_MB_DATA3(8));
  W[P0] = R0;  /* Byte0 = 0xAA, Byte1 = 0xBB */

  R0 = 1;
  P0.L = LO(CAN0_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(CAN0_MB_ID1(9));  /* Access MB08 ID1 Register */
  W[P0] = R0;  /* Remote frame disabled, 11 bit ID */
```

```
R0 = 0;
P0.L = LO(CAN0_MB_ID0(9));
W[P0] = R0;  /* Zero Out Lower ID Register */
SSYNC;

/* Enable the Configured Mailboxes */
P0.L = LO(CAN0_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 `CAN0_RX_HANDLER` and `CAN0_TX_HANDLER` have been properly registered in the system interrupt controller and that the interrupts are enabled properly in the `SIC_IMASK0` register.

Listing 31-4. CAN Transfers and Interrupts

```
CAN0_SetupIRQs_and_Transfer:
  P0.H = HI(CAN0_MBIM1);
  P0.L = LO(CAN0_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(CAN0_CONTROL);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(CCR));
  W[P0] = R0;

  P0.L = LO(CAN0_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(CAN0_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;
/* ================================================
** CAN0_TX_HANDLER
**
** ISR clears the interrupt request from MB8, writes
** new data to be sent, and requests to send again
**
** ================================================
*/
CAN0_TX_HANDLER:
  [--SP] = (R7:6, P5:5);  /* Save Clobbered Registers */
  [--SP] = ASTAT;
  P5.H = HI(CAN0_MB_DATA2(8));
```

```
  P5.L = LO(CAN0_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(CAN0_TRS1);
    R7 = TRS8;
    W[P5] = R7;  /* Issue New Transmit Request */
    P5.L = LO(CAN0_MBTIF1);
    R7 = MBTIF8;
    W[P5] = R7; /* Clear Interrupt Request Bit for MB08 */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:6, P5:5) = [SP++];
  SSYNC;
  RTI;
/* =================================================
** CAN0_RX_HANDLER
**
** ISR clears the interrupt request from MB9, writes
** new data to be sent, and requests to send again
**
** =================================================*/
```

```
CAN0_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(CAN0_MBRMP1);
  P5.L = LO(CAN0_MBRMP1);
  R7 = RMP9;
  W[P5] = R7; /* Clear Message Pending for MB09 */
  P5.L = LO(CANx_MBRIF1);
  R7 = MBRIF9;
  W[P5] = R7;  /* Clear Interrupt Request Bit for MB09 */
  P5.L = LO(CAN_RMP1);
  W[P5] = R7; /* Clear Message Pending Bit for MB09 */
  P5.L = LO(CAN0_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;
```

A SYSTEM MMR ASSIGNMENTS

This appendix lists MMR addresses and register names for the system memory-mapped registers (MMRs), the core timer registers, and the processor-specific memory registers mentioned in this manual, To find more information about an MMR, refer to the page shown in the “See Page” column. When viewing the PDF version of this document, click a reference in the “See Page” column to jump to additional information about the MMR.

This chapter includes the following sections:

“Dynamic Power Management Registers” on page A-4

“System Reset and Interrupt Control Registers” on page A-4

“Watchdog Timer Registers” on page A-6

“Real-Time Clock Registers” on page A-6

“UART0 Controller Registers” on page A-7

“UART1 Controller Registers” on page A-8

“UART2 Controller Registers” on page A-9

“UART3 Controller Registers” on page A-10

“SPI0 Controller Registers” on page A-11

“SPI1 Controller Registers” on page A-11

“SPI2 Controller Registers” on page A-12

"TWI0 Registers" on page A-13

"TWI1 Registers" on page A-14

"SPORT0 Controller Registers" on page A-16

"SPORT1 Controller Registers" on page A-18

"SPORT2 Controller Registers" on page A-20

"SPORT3 Controller Registers" on page A-22

"MXVR Registers" on page A-24

"Keypad Registers" on page A-36

"SDH Registers" on page A-37

"ATAPI Registers" on page A-39

"USB OTG Registers" on page A-41

"External Bus Interface Unit Registers" on page A-58

"DMA/Memory DMA Control Registers" on page A-59

"EPPI0 Registers" on page A-62

"EPPI1 Registers" on page A-63

"EPPI2 Registers" on page A-64

"Host DMA Registers" on page A-65

"PIXC Registers" on page A-66

"Ports Registers" on page A-68

"Timer Registers" on page A-76

"CANx Registers" on page A-79

"Handshake MDMA Control Registers" on page A-88

"NAND Flash Controller Registers" on page A-90

"Core Timer Registers" on page A-91

"Rotary Counter Registers" on page A-91

"Security Registers" on page A-92

"Processor-Specific Memory Registers" on page A-93

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	Description
0xFFC0 0000	PLL_CTL	"PLL Control (PLL_CTL) Register" on page 18-26
0xFFC0 0004	PLL_DIV	"PLL Divide (PLL_DIV) Register" on page 18-26
0xFFC0 0008	VR_CTL	"Voltage Regulator Control (VR_CTL) Register" on page 18-28
0xFFC0 000C	PLL_STAT	"PLL Status (PLL_STAT) Register" on page 18-27
0xFFC0 0010	PLL_ LOCKCNT	"PLL Lock Count (PLL_LOCKCNT) Register" on page 18-27

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	Description
0xFFC0 0100	SYRST	"Software Reset (SWRST) Register" on page 17-103
0xFFC0 0104	SYSCR	"System Reset Configuration (SYSCR) Register" on page 17-105
0xFFC0 010C	SIC_IMASK0	"System Interrupt Mask Register 0" on page 6-32

Table A-2. System Reset and Interrupt Control Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 0110	SIC_IMASK1	"System Interrupt Mask Register 1" on page 6-33
0xFFC0 0114	SIC_IMASK2	"System Interrupt Mask Register 2" on page 6-34
0xFFC0 0130	SIC_IAR0	"System Interrupt Assignment Register 0" on page 6-26
0xFFC0 0134	SIC_IAR1	"System Interrupt Assignment Register 1" on page 6-26
0xFFC0 0138	SIC_IAR2	"System Interrupt Assignment Register 2" on page 6-27
0xFFC0 013C	SIC_IAR3	"System Interrupt Assignment Register 3" on page 6-27
0xFFC0 0140	SIC_IAR4	"System Interrupt Assignment Register 4" on page 6-28
0xFFC0 0144	SIC_IAR5	"System Interrupt Assignment Register 5" on page 6-28
0xFFC0 0148	SIC_IAR6	"System Interrupt Assignment Register 6" on page 6-29
0xFFC0 014C	SIC_IAR7	"System Interrupt Assignment Register 7" on page 6-29
0xFFC0 0150	SIC_IAR8	"System Interrupt Assignment Register 8" on page 6-30
0xFFC0 0154	SIC_IAR9	"System Interrupt Assignment Register 9" on page 6-30
0xFFC0 0158	SIC_IAR10	"System Interrupt Assignment Register 10" on page 6-31
0xFFC0 015C	SIC_IAR11	"System Interrupt Assignment Register 11" on page 6-31
0xFFC0 0118	SIC_ISR0	"System Interrupt Status Register 0" on page 6-35
0xFFC0 011C	SIC_ISR1	"System Interrupt Status Register 1" on page 6-36
0xFFC0 0120	SIC_ISR2	"System Interrupt Status Register 2" on page 6-37
0xFFC0 0124	SIC_IWR0	"System Interrupt Wakeup Register 0" on page 6-38
0xFFC0 0128	SIC_IWR1	"System Interrupt Wakeup Register 1" on page 6-39
0xFFC0 012C	SIC_IWR2	"System Interrupt Wakeup Register 2" on page 6-40

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	Description
0xFFC0 0200	WDOG_CTL	"Watchdog Control (WDOG_CTL) Register" on page 12-8
0xFFC0 0204	WDOG_CNT	"Watchdog Count (WDOG_CNT) Register" on page 12-6
0xFFC0 0208	WDOG_ STAT	"Watchdog Status (WDOG_STAT) Register" on page 12-7

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	Description
0xFFC0 0300	RTC_STAT	"RTC Status (RTC_STAT) Register" on page 14-21
0xFFC0 0304	RTC_ICTL	"RTC Interrupt Control (RTC_ICTL) Register" on page 14-21
0xFFC0 0308	RTC_ISTAT	"RTC Interrupt Status (RTC_ISTAT) Register" on page 14-22
0xFFC0 030C	RTC_ SWCNT	"RTC Stopwatch Count (RTC_SWCNT) Register" on page 14-22
0xFFC0 0310	RTC_ALARM	"RTC Alarm (RTC_ALARM) Register" on page 14-23
0xFFC0 0314	RTC_PREN	"RTC Prescaler Enable (RTC_PREN) Register" on page 14-23

UART0 Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in Table A-5.

Table A-5. UART0 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0400	UART0_DLL	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 0404	UART0_DLH	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 0408	UART0_GCTL	"Global Control (UARTx_GCTL) Registers" on page 25-50
0xFFC0 040C	UART0_LCR	"Line Control (UARTx_LCR) Registers" on page 25-29
0xFFC0 0410	UART0_MCR	"Modem Control (UARTx_MCR) Registers" on page 25-32
0xFFC0 0414	UART0_LSR	"Line Status (UARTx_LSR) Registers" on page 25-34
0xFFC0 0418	UART0_MSR	"Modem Status (UARTx_MSR) Registers" on page 25-37
0xFFC0 041C	UART0_SCR	"UART Scratch (UARTx_SCR) Registers" on page 25-49
0xFFC0 0420	UART0_IER_SET	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 0424	UART0_IER_CLEAR	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 0428	UART0_THR	"Transmit Hold (UARTx_THR) Registers" on page 25-39
0xFFC0 042C	UART0_RBR	"Receive Buffer (UARTx_RBR) Registers" on page 25-40

UART1 Controller Registers

UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in Table A-6.

Table A-6. UART1 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0200	UART1_DLL	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 0204	UART1_DLH	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 0208	UART1_GCTL	"Global Control (UARTx_GCTL) Registers" on page 25-50
0xFFC0 020C	UART1_LCR	"Line Control (UARTx_LCR) Registers" on page 25-29
0xFFC0 0210	UART1_MCR	"Modem Control (UARTx_MCR) Registers" on page 25-32
0xFFC0 0214	UART1_LSR	"Line Status (UARTx_LSR) Registers" on page 25-34
0xFFC0 0218	UART1_MSR	"Modem Status (UARTx_MSR) Registers" on page 25-37
0xFFC0 021C	UART1_SCR	"UART Scratch (UARTx_SCR) Registers" on page 25-49
0xFFC0 0220	UART1_IER_SET	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 0224	UART1_IER_CLEAR	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 0228	UART1_THR	"Transmit Hold (UARTx_THR) Registers" on page 25-39
0xFFC0 022C	UART1_RBR	"Receive Buffer (UARTx_RBR) Registers" on page 25-40

UART2 Controller Registers

UART2 controller registers are listed in Table A-7. UART2 is not available on the ADSP-BF542 and ADSP-BF544 processors.

Table A-7. UART2 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2100	UART2_DLL	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 2104	UART2_DLH	"Clock Divisor Latch (UARTx_DLL and UARTx_DLH) Registers" on page 25-46
0xFFC0 2108	UART2_GCTL	"Global Control (UARTx_GCTL) Registers" on page 25-50
0xFFC0 210C	UART2_LCR	"Line Control (UARTx_LCR) Registers" on page 25-29
0xFFC0 2110	UART2_MCR	"Modem Control (UARTx_MCR) Registers" on page 25-32
0xFFC0 2114	UART2_LSR	"Line Status (UARTx_LSR) Registers" on page 25-34
0xFFC0 2118	UART2_MSR	"Modem Status (UARTx_MSR) Registers" on page 25-37
0xFFC0 211C	UART2_SCR	"UART Scratch (UARTx_SCR) Registers" on page 25-49
0xFFC0 2120	UART2_IER_SET	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 2124	UART2_IER_CLEAR	"Interrupt Enable (UARTx_IER_SET and UARTx_IER_CLEAR) Registers" on page 25-40
0xFFC0 2128	UART2_THR	"Transmit Hold (UARTx_THR) Registers" on page 25-39
0xFFC0 212C	UART2_RBR	"Receive Buffer (UARTx_RBR) Registers" on page 25-40

UART3 Controller Registers

UART3 controller registers are listed in Table A-8.

Table A-8. UART3 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3100	UART3_DLL	"UART Divisor Latch Registers" on page 25-48
0xFFC0 3104	UART3_DLH	"UART Divisor Latch Registers" on page 25-48
0xFFC0 3108	UART3_GCTL	"UART Global Control Registers" on page 25-50
0xFFC0 310C	UART3_LCR	"UART Line Control Registers" on page 25-29
0xFFC0 3110	UART3_MCR	"UART Modem Control Registers" on page 25-32
0xFFC0 3114	UART3_LSR	"UART Line Status Registers" on page 25-35
0xFFC0 3118	UART3_MSR	"UART Modem Control Registers" on page 25-32
0xFFC0 311C	UART3_SCR	"UART Scratch Registers" on page 25-49
0xFFC0 3120	UART3_IER_SET	"UART Interrupt Enable Set Registers" on page 25-42
0xFFC0 3124	UART3_IER_ CLEAR	"UART Interrupt Enable Clear Registers" on page 25-43
0xFFC0 3128	UART3_THR	"UART Transmit Holding Registers" on page 25-39
0xFFC0 312C	UART3_RBR	"UART Receive Buffer Registers" on page 25-40

SPI0 Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in Table A-9.

Table A-9. SPI0 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0500	SPI0_CTL	"SPI Control (SPIx_CTL) Register" on page 22-45
0xFFC0 0504	SPI0_FLG	"SPI Flag (SPIx_FLG) Register" on page 22-46
0xFFC0 0508	SPI0_STAT	"SPI Status (SPIx_STAT) Register" on page 22-48
0xFFC0 050C	SPI0_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 22-48
0xFFC0 0510	SPI0_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 22-49
0xFFC0 0514	SPI0_BAUD	"SPI Baud Rate (SPIx_BAUD) Register" on page 22-44
0xFFC0 0518	SPI0_ SHADOW	"SPI RDBR Shadow (SPIx_SHADOW) Register" on page 22-49

SPI1 Controller Registers

SPI1 controller registers are listed in Table A-10.

Table A-10. SPI1 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2300	SPI1_CTL	"SPI Control (SPIx_CTL) Register" on page 22-45
0xFFC0 2304	SPI1_FLG	"SPI Flag (SPIx_FLG) Register" on page 22-46
0xFFC0 2308	SPI1_STAT	"SPI Status (SPIx_STAT) Register" on page 22-48

Table A-10. SPI1 Controller Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 230C	SPI1_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 22-48
0xFFC0 2310	SPI1_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 22-49
0xFFC0 2314	SPI1_BAUD	"SPI Baud Rate (SPIx_BAUD) Register" on page 22-44
0xFFC0 2318	SPI1_ SHADOW	"SPI RDBR Shadow (SPIx_SHADOW) Register" on page 22-49

SPI2 Controller Registers

SPI2 controller registers are listed in Table A-10.

Table A-11. SPI2 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2400	SPI2_CTL	"SPI Control (SPIx_CTL) Register" on page 22-45
0xFFC0 2404	SPI2_FLG	"SPI Flag (SPIx_FLG) Register" on page 22-46
0xFFC0 2408	SPI2_STAT	"SPI Status (SPIx_STAT) Register" on page 22-48
0xFFC0 240C	SPI2_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 22-48
0xFFC0 2410	SPI2_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 22-49
0xFFC0 2414	SPI2_BAUD	"SPI Baud Rate (SPIx_BAUD) Register" on page 22-44
0xFFC0 2418	SPI2_ SHADOW	"SPI RDBR Shadow (SPIx_SHADOW) Register" on page 22-49

TWI0 Registers

The TWI0 controller has 16 registers described in the following sections.

Table A-13 lists the TWI0 registers.

Table A-12. TWI0 Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0700	TWI0_CLKDIV	"SCLx Clock Divider (TWIx_CLKDIV) Register" on page 23-26
0xFFC0 0704	TWI0_CONTROL	"TWI Control (TWIx_CONTROL) Register" on page 23-27
0xFFC0 0708	TWI0_SLAVE_CTL	"TWI Slave Mode Control (TWIx_SLAVE_CTL) Register" on page 23-27
0xFFC0 0710	TWI0_SLAVE_ADDR	"TWI Slave Mode Address (TWIx_SLAVE_ADDR) Register" on page 23-30
0xFFC0 070C	TWI0_SLAVE_STAT	"TWI Slave Mode Status (TWIx_SLAVE_STAT) Register" on page 23-30
0xFFC0 0714	TWI0_MASTER_CTL	"TWI Master Mode Control (TWIx_MASTER_CTL) Register" on page 23-32
0xFFC0 071C	TWI0_MASTER_ ADDR	"TWI Master Mode Address (TWIx_MASTER_ ADDR) Register" on page 23-35
0xFFC0 0718	TWI0_MASTER_STAT	"TWI Master Mode Status (TWIx_MASTER_STAT) Register" on page 23-35
0xFFC0 0720	TWI0_INT_STAT	"TWI Interrupt Status (TWIx_INT_STAT) Register" on page 23-44
0xFFC0 0724	TWI0_INT_MASK	"TWI Interrupt Mask (TWIx_INT_MASK) Register" on page 23-43
0xFFC0 0728	TWI0_FIFO_CTL	"TWI FIFO Control (TWIx_FIFO_CTL) Register" on page 23-39
0xFFC0 072C	TWI0_FIFO_STAT	"TWI FIFO Status (TWIx_FIFO_STAT) Register" on page 23-41

Table A-12. TWI0 Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 0780	TWI0_XMT_DATA8	"TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Register" on page 23-48
0xFFC0 0784	TWI0_XMT_DATA16	"TWI FIFO Transmit Data Double Byte Register" on page 23-49
0xFFC0 0788	TWI0_RCV_DATA8	"TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Register" on page 23-50
0xFFC0 078C	TWI0_RCV_DATA16	"TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Register" on page 23-51

TWI1 Registers

The TWI1 controller has 16 registers described in the following sections.

Table A-13 lists the TW1I registers.

Table A-13. TWI1 Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2200	TWI1_CLKDIV	"SCLx Clock Divider (TWIx_CLKDIV) Register" on page 23-26
0xFFC0 2204	TWI1_CONTROL	"TWI Control (TWIx_CONTROL) Register" on page 23-27
0xFFC0 2208	TWI1_SLAVE_CTL	"TWI Slave Mode Control (TWIx_SLAVE_CTL) Register" on page 23-27
0xFFC0 2210	TWI1_SLAVE_ADDR	"TWI Slave Mode Address (TWIx_SLAVE_ADDR) Register" on page 23-30
0xFFC0 220C	TWI1_SLAVE_STAT	"TWI Slave Mode Status (TWIx_SLAVE_STAT) Register" on page 23-30

Table A-13. TWI1 Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2214	TWI1_MASTER_CTL	"TWI Master Mode Control (TWIx_MASTER_CTL) Register" on page 23-32
0xFFC 221C	TWI1_MASTER_ ADDR	"TWI Master Mode Address (TWIx_MASTER_ ADDR) Register" on page 23-35
0xFFC0 2218	TWI1_MASTER_STAT	"TWI Master Mode Status (TWIx_MASTER_STAT) Register" on page 23-35
0xFFC0 2220	TWI1_INT_STAT	"TWI Interrupt Status (TWIx_INT_STAT) Register" on page 23-44
0xFFC0 2224	TWI1_INT_MASK	"TWI Interrupt Mask (TWIx_INT_MASK) Register" on page 23-43
0xFFC0 2228	TWI1_FIFO_CTL	"TWI FIFO Control (TWIx_FIFO_CTL) Register" on page 23-39
0xFFC0 222C	TWI1_FIFO_STAT	"TWI FIFO Status (TWIx_FIFO_STAT) Register" on page 23-41
0xFFC0 2280	TWI1_XMT_DATA8	"TWI FIFO Transmit Data Single Byte (TWIx_XMT_ DATA8) Register" on page 23-48
0xFFC0 2284	TWI1_XMT_DATA16	"TWI FIFO Transmit Data Double Byte Register" on page 23-49
0xFFC0 2288	TWI1_RCV_DATA8	"TWI FIFO Receive Data Single Byte (TWIx_RCV_ DATA8) Register" on page 23-50
0xFFC0 228C	TWI1_RCV_DATA16	"TWI FIFO Receive Data Double Byte (TWIx_RCV_ DATA16) Register" on page 23-51

SPORT0 Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in Table A-14.

Table A-14. SPORT0 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0800	SPORT0_TCR1	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 0804	SPORT0_TCR2	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 0808	SPORT0_TCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 080C	SPORT0_TFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 0810	SPORT0_TX	"Transmit Data (SPORTx_TX) Register" on page 24-61
0xFFC0 0818	SPORT0_RX	"Receive Data (SPORTx_RX) Register" on page 24-64
0xFFC0 0820	SPORT0_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 0824	SPORT0_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 0828	SPORT0_RCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 082C	SPORT0_RFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 0830	SPORT0_STAT	"SPORT Status (SPORTx_STAT) Register" on page 24-66
0xFFC0 0834	SPORT0_CHNL	"Current Channel (SPORTx_CHNL) Register" on page 24-71
0xFFC0 0838	SPORT0_MCMC1	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70

Table A-14. SPORT0 Controller Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 083C	SPORT0_MCMC2	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70
0xFFC0 0840	SPORT0_MTCS0	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0844	SPORT0_MTCS1	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0848	SPORT0_MTCS2	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 084C	SPORT0_MTCS3	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0850	SPORT0_MRCS0	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 0854	SPORT0_MRCS1	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 0858	SPORT0_MRCS2	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 085C	SPORT0_MRCS3	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72

SPORT1 Controller Registers

SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-15.

Table A-15. SPORT 1 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0900	SPORT1_TCR1	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 0904	SPORT1_TCR2	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 0908	SPORT1_TCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 090C	SPORT1_TFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 0910	SPORT1_TX	"Transmit Data (SPORTx_TX) Register" on page 24-61
0xFFC0 0918	SPORT1_RX	"Receive Data (SPORTx_RX) Register" on page 24-64
0xFFC0 0920	SPORT1_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 0924	SPORT1_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 0928	SPORT1_RCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 092C	SPORT1_RFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 0930	SPORT1_STAT	"SPORT Status (SPORTx_STAT) Register" on page 24-66
0xFFC0 0934	SPORT1_CHNL	"Current Channel (SPORTx_CHNL) Register" on page 24-71
0xFFC0 0938	SPORT1_MCMC1	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70

Table A-15. SPORT 1 Controller Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 093C	SPORT1_MCMC2	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70
0xFFC0 0940	SPORT1_MTCS0	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0944	SPORT1_MTCS1	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0948	SPORT1_MTCS2	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 094C	SPORT1_MTCS3	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 0950	SPORT1_MRCS0	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 0954	SPORT1_MRCS1	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 0958	SPORT1_MRCS2	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 095C	SPORT1_MRCS3	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72

SPORT2 Controller Registers

SPORT2 controller registers are listed in Table A-16.

Table A-16. SPORT2 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2500	SPORT2_TCR1	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 2504	SPORT2_TCR2	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 2508	SPORT2_TCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 250C	SPORT2_TFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 2510	SPORT2_TX	"Transmit Data (SPORTx_TX) Register" on page 24-61
0xFFC0 2518	SPORT2_RX	"Receive Data (SPORTx_RX) Register" on page 24-64
0xFFC0 2520	SPORT2_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 2524	SPORT2_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 2528	SPORT2_RCLK-DIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 252C	SPORT2_RFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 2530	SPORT2_STAT	"SPORT Status (SPORTx_STAT) Register" on page 24-66
0xFFC0 2534	SPORT2_CHNL	"Current Channel (SPORTx_CHNL) Register" on page 24-71
0xFFC0 2538	SPORT2_MCMC1	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70
0xFFC0 253C	SPORT2_MCMC2	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70

Table A-16. SPORT2 Controller Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2540	SPORT2_MTCS0	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2544	SPORT2_MTCS1	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2548	SPORT2_MTCS2	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 254C	SPORT2_MTCS3	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2550	SPORT2_MRCS0	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 2554	SPORT2_MRCS1	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 2558	SPORT2_MRCS2	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 255C	SPORT2_MRCS3	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72

SPORT3 Controller Registers

SPORT3 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-17.

Table A-17. SPORT3 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2600	SPORT3_TCR1	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 2604	SPORT3_TCR2	"Transmit Configuration (SPORTx_TCR1 and SPORTx_TCR2) Registers" on page 24-51
0xFFC0 2608	SPORT3_TCLKDIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 260C	SPORT3_TFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 2610	SPORT3_TX	"Transmit Data (SPORTx_TX) Register" on page 24-61
0xFFC0 2618	SPORT3_RX	"Receive Data (SPORTx_RX) Register" on page 24-64
0xFFC0 2620	SPORT3_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 2624	SPORT3_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 24-56
0xFFC0 2628	SPORT3_RCLKDIV	"Serial Clock Divider (SPORTx_TCLKDIV and SPORTx_RCLKDIV) Registers" on page 24-68
0xFFC0 262C	SPORT3_RFSDIV	"Frame Sync Divider (SPORTx_TFSDIV and SPORTx_RFSDIV) Registers" on page 24-69
0xFFC0 2630	SPORT3_STAT	"SPORT Status (SPORTx_STAT) Register" on page 24-66
0xFFC0 2634	SPORT3_CHNL	"Current Channel (SPORTx_CHNL) Register" on page 24-71
0xFFC0 2638	SPORT3_MCMC1	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70

Table A-17. SPORT3 Controller Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 263C	SPORT3_MCMC2	"Multichannel Configuration (SPORTx_MCMCn) Registers" on page 24-70
0xFFC0 2640	SPORT3_MTCS0	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2644	SPORT3_MTCS1	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2648	SPORT3_MTCS2	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 264C	SPORT3_MTCS3	"Multichannel Selection Transmit (SPORTx_MTCSn) Registers" on page 24-74
0xFFC0 2650	SPORT3_MRCS0	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 2654	SPORT3_MRCS1	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 2658	SPORT3_MRCS2	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72
0xFFC0 265C	SPORT3_MRCS3	"Multichannel Selection Receive (SPORTx_MRCSn) Registers" on page 24-72

MXVR Registers

Table A-18 shows the MXVR register memory map.

Table A-18. MXVR Memory Map

Memory Mapped Address	Register Name	Description
0xFFC0 2700	MXVR_CONFIG 16-bit R/W Reset = 0x1FCA	"MXVR Configuration (MXVR_CONFIG) Register" on page 29-13
0xFFC0 2704	Reserved	–
0xFFC0 2708	MXVR_STATE_0 32-bit RO Reset = 0x0000 0000	"MXVR State (MXVR_STATE_0, MXVR_STATE_1) Registers" on page 29-19
0xFFC0 270C	MXVR_STATE_1 32-bit RO Reset = 0x0000 0000	"MXVR State (MXVR_STATE_0, MXVR_STATE_1) Registers" on page 29-19
0xFFC0 2710	MXVR_INT_STAT_0 32-bit R?W Reset = 0x0000 0000	"MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)" on page 29-29
0xFFC0 2714	MXVR_INT_STAT_1 32-bit R/W Reset = 0x0000 0000	"MXVR Interrupt Status_1 (MXVR_INT_STAT_1) Register" on page 29-40
0xFFC0 2718	MXVR_INT_EN_0 32-bit R/W Reset = 0x0000 0000	"MXVR Interrupt Enable 0 (MXVR_INT_EN_0) Register" on page 29-43
0xFFC0 271C	MXVR_INT_EN_1 32-bit R/W Reset = 0x0000 0000	"MXVR Interrupt Enable 1 (MXVR_INT_EN_1) Register" on page 29-46
0xFFC0 2720	MXVR_POSITION 16-bit RO Reset = 0x8000	"MXVR Node Position (MXVR_POSITION) Register" on page 29-48
0xFFC0 2724	MXVR_MAX_POSITION 16-bit RO Reset = 0x0000	"MXVR Maximum Node Position (MXVR_MAX_POSITION) Register" on page 29-49

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2728	MXVR_DELAY 16-bit RO Reset = 0x8000	"MXVR Node Frame Delay (MXVR_DELAY) Register" on page 29-50
0xFFC0 272C	MXVR_MAX_DELAY 16-bit RO Reset = 0x0000	"MXVR Maximum Node Frame Delay (MXVR_MAX_DELAY) Register" on page 29-52
0xFFC0 2730	MXVR_LADDR 32-bit R/W Reset = 0x0000 0FFF	"MXVR Logical Address (MXVR_LADDR) Register" on page 29-53
0xFFC0 2734	MXVR_GADDR 16-bit R/W Reset = 0x0000	"MXVR Group Address (MXVR_GADDR) Register" on page 29-54
0xFFC0 2738	MXVR_AADDR 32-bit R/W Reset = 0x0000 0FFF	"MXVR Alternate Address (MXVR_AADDR) Register" on page 29-55
0xFFC0 273C	MXVR_ALLOC_0 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2740	MXVR_ALLOC_1 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2744	MXVR_ALLOC_2 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2748	MXVR_ALLOC_3 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 274C	MXVR_ALLOC_4 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2750	MXVR_ALLOC_5 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2754	MXVR_ALLOC_6 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2758	MXVR_ALLOC_7 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 275C	MXVR_ALLOC_8 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2760	MXVR_ALLOC_9 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2764	MXVR_ALLOC_10 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2768	MXVR_ALLOC_11 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 276C	MXVR_ALLOC_12 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2770	MXVR_ALLOC_13 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55
0xFFC0 2774	MXVR_ALLOC_14 32-bit RO Reset = 0xXXXX XXXX	"MXVR Allocation Table (MXVR_ALLOC_0 – MXVR_ALLOC_14) Registers" on page 29-55

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2778	MXVR_SYNC_LCHAN_0 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 277C	MXVR_SYNC_LCHAN_1 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2780	MXVR_SYNC_LCHAN_2 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2784	MXVR_SYNC_LCHAN_3 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2788	MXVR_SYNC_LCHAN_4 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 278C	MXVR_SYNC_LCHAN_5 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2790	MXVR_SYNC_LCHAN_6 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2794	MXVR_SYNC_LCHAN_7 32-bit R/W Reset = 0xFFFF FFFF	"MXVR Synchronous Logical Channel Assignment (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7) Registers" on page 29-57
0xFFC0 2798	MXVR_DMA0_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59
0xFFC0 279C	MXVR_DMA0_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 27A0	MXVR_DMA0_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 27A4	MXVR_DMA0_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 27A8	MXVR_DMA0_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 27AC	MXVR_DMA1_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59
0xFFC0 27B0	MXVR_DMA1_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 27B4	MXVR_DMA1_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 27B8	MXVR_DMA1_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 27BC	MXVR_DMA1_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 27C0	MXVR_DMA2_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 27C4	MXVR_DMA2_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 27C8	MXVR_DMA2_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 27CC	MXVR_DMA2_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 27D0	MXVR_DMA2_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 27D4	MXVR_DMA3_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59
0xFFC0 27D8	MXVR_DMA3_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 27DC	MXVR_DMA3_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 27E0	MXVR_DMA3_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 27E4	MXVR_DMA3_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 27E8	MXVR_DMA4_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59
0xFFC0 27EC	MXVR_DMA4_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 27F0	MXVR_DMA4_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 27F4	MXVR_DMA4_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 27F8	MXVR_DMA4_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 27FC	MXVR_DMA5_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG) Registers" on page 29-59
0xFFC0 2800	MXVR_DMA5_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 2804	MXVR_DMA5_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT) Registers" on page 29-72
0xFFC0 2808	MXVR_DMA5_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 280C	MXVR_DMA5_CURR_ COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 2810	MXVR_DMA6_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_ DMA0_CONFIG – MXVR_DMA7_CON- FIG) Registers" on page 29-59
0xFFC0 2814	MXVR_DMA6_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_ DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 2818	MXVR_DMA6_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_ COUNT) Registers" on page 29-72
0xFFC0 281C	MXVR_DMA6_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_ DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 2820	MXVR_DMA6_CURR_ COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 2824	MXVR_DMA7_CONFIG 32-bit R/W Reset = 0x0000 0000	"MXVR DMAx Configuration (MXVR_ DMA0_CONFIG – MXVR_DMA7_CON- FIG) Registers" on page 29-59
0xFFC0 2828	MXVR_DMA7_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR DMA Channel x Start Address (MXVR_DMA0_START_ADDR – MXVR_ DMA7_START_ADDR) Registers" on page 29-69
0xFFC0 282C	MXVR_DMA7_COUNT 16-bit R/W Reset = 0x0001	"MXVR DMA Channel x Transfer Count (MXVR_DMA0_COUNT – MXVR_DMA7_ COUNT) Registers" on page 29-72

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2830	MXVR_DMA7_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR DMA Channel x Current Address (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR) Registers" on page 29-71
0xFFC0 2834	MXVR_DMA7_CURR_COUNT 16-bit RO Reset = 0x0000	"MXVR DMA Channel x Current Transfer Count (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT) Registers" on page 29-74
0xFFC0 2838	MXVR_AP_CTL 16-bit R/W Reset = 0x0000	"MXVR Asynchronous Packet Control (MXVR_AP_CTL) Register" on page 29-75
0xFFC0 283C	MXVR_APRB_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR Asynchronous Packet Receive Buffer Start Address (MXVR_APRB_START_ADDR) Register" on page 29-77
0xFFC0 2840	MXVR_APRB_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR Asynchronous Packet Receive Buffer Current Address (MXVR_APRB_CURR_ADDR) Register" on page 29-78
0xFFC0 2844	MXVR_APTB_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR Asynchronous Packet Transmit Buffer Start Address (MXVR_APTB_START_ADDR) Register" on page 29-79
0xFFC0 2848	MXVR_APTB_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR Asynchronous Packet Transmit Buffer Current Address (MXVR_APTB_CURR_ADDR) Register" on page 29-79
0xFFC0 284C	MXVR_CM_CTL 32-bit R/W Reset = 0x0000 0000	"MXVR Control Message Control (MXVR_CM_CTL) Register" on page 29-80
0xFFC0 2850	MXVR_CMRB_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR Control Message Receive Buffer Start Address (MXVR_CMRB_START_ADDR) Register" on page 29-82
0xFFC0 2854	MXVR_CMRB_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR Control Message Receive Buffer Start Address (MXVR_CMRB_START_ADDR) Register" on page 29-82

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2858	MXVR_CMTB_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR Control Message Transmit Buffer Start Address (MXVR_CMTB_START_ADDR) Register" on page 29-84
0xFFC0 285C	MXVR_CMTB_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR Control Message Transmit Buffer Current Address (MXVR_CMTB_CURR_ADDR) Register" on page 29-85
0xFFC0 2860	MXVR_RRDB_START_ADDR 32-bit R/W Reset = 0xFF00 0000	"MXVR Remote Read Buffer Start Address (MXVR_RRDB_START_ADDR) Register" on page 29-86
0xFFC0 2864	MXVR_RRDB_CURR_ADDR 32-bit RO Reset = 0xFF00 0000	"MXVR Remote Read Buffer Current Address (MXVR_RRDB_CURR_ADDR) Register" on page 29-86
0xFFC0 2868	MXVR_PAT_DATA_0 32-bit R/W Reset = 0x0000 0000	"MXVR Pattern Data (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1) Registers" on page 29-87
0xFFC0 286C	MXVR_PAT_EN_0 32-bit R/W Reset = 0x0000 0000	"MXVR Pattern Enable (MXVR_PAT_EN_0, MXVR_PAT_EN_1) Registers" on page 29-88
0xFFC0 2870	MXVR_PAT_DATA_1 32-bit R/W Reset = 0x0000 0000	"MXVR Pattern Data (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1) Registers" on page 29-87
0xFFC0 2874	MXVR_PAT_EN_1 32-bit R/W Reset = 0x0000 0000	"MXVR Pattern Enable (MXVR_PAT_EN_0, MXVR_PAT_EN_1) Registers" on page 29-88
0xFFC0 2878	MXVR_FRAME_CNT_0 16-bit R/W Reset = 0x0000	"MXVR Frame Counter (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1) Registers" on page 29-90
0xFFC0 287C	MXVR_FRAME_CNT_1 16-bit R/W Reset = 0x0000	"MXVR Frame Counter (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1) Registers" on page 29-90

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2880	MXVR_Routing_0 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 2884	MXVR_Routing_1 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 2888	MXVR_Routing_2 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 288C	MXVR_Routing_3 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 2890	MXVR_Routing_4 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 2894	MXVR_Routing_5 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 2898	MXVR_Routing_6 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 289C	MXVR_Routing_7 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28A0	MXVR_Routing_8 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28A4	MXVR_Routing_9 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 28A8	MXVR_Routing_10 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28AC	MXVR_Routing_11 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28B0	MXVR_Routing_12 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28B4	MXVR_Routing_13 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28B8	MXVR_Routing_14 32-bit WO Reset = 0xXXXX XXXX	"MXVR Routing (MXVR_ROUTING_0 – MXVR_ROUTING_14) Registers" on page 29-91
0xFFC0 28BC	Reserved	–
0xFFC0 28C0	MXVR_BLOCK_CNT 16-bit R/W Reset = 0x0000	"MXVR Block Counter (MXVR_BLOCK_CNT) Register" on page 29-94
0xFFC0 28C4 to 0xFFC0 28CC	Reserved	–
0xFFC0 28D0	MXVR_CLK_CTL 32-bit R/W Reset = 0x0202 0003	"MXVR Clock Control (MXVR_CLK_CTL) Register" on page 29-95
0xFFC0 28D4	MXVR_CDRPLL_CTL 32-bit R/W Reset = 0x0502 0820	"MXVR Clock/Data Recovery PLL Control (MXVR_CDRPLL_CTL) Register" on page 29-101
0xFFC0 28D8	MXVR_FMPLL_CTL 32-bit R/W Reset = 0x1900 1020	"MXVR Frequency Multiply PLL Control (MXVR_FMPLL_CTL) Register" on page 29-104

Table A-18. MXVR Memory Map (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 28DC	MXVR_PIN_CTL 16-bit R/W Reset = 0x0000	"MXVR Pin Control (MXVR_PIN_CTL) Register" on page 29-106
0xFFC0 28E0	MXVR_SCLK_CNT 16-bit R/W Reset = 0x0000	"MXVR System Clock Counter (MXVR_SCLK_CNT) Register" on page 29-107
0xFFC0 28E4 to 0xFFC0 28FF	Reserved	–

Keypad Registers

Descriptions and bit diagrams for each of the memory-mapped registers (MMRs) are provided in the following subsections. See Table A-19.

Table A-19. Control/Status/Data Registers

Memory Mapped Address	Register Name	Description
0xFFC0 4100	KPAD_CTL	"Keypad Control (KPAD_CTL) Register" on page 30-10
0xFFC0 4104	KPAD_PRESCALE	"Keypad Prescale (KPAD_PRESCALE) Register" on page 30-13
0xFFC0 4108	KPAD_MSEL	"Keypad Multiplier Select (KPAD_MSEL) Register" on page 30-15
0xFFC0 410C	KPAD_ROWCOL	"Keypad Row-Column (KPAD_ROWCOL) Register" on page 30-15
0xFFC0 4110	KPAD_STAT	"Keypad Status (KPAD_STAT) Register" on page 30-18
0xFFC0 4114	KPAD_SOFTE-VAL	"Keypad Software Evaluate (KPAD_SOFTEVAL) Register" on page 30-20

SDH Registers

The Secure Data Host (SDH) interface has memory-mapped registers (MMRs) that regulate its operation. Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

The SDH memory-mapped registers start at base address 0xFFC03900. In Table A-20, register addresses are given relative to the base address. All functional register bits reset to zero, *except* the SDH_E_MASKx registers (which reset to 0x40) and SDH_CFG register (which resets to 0xA0).

Table A-20. SDH Functional Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3900	SDH_PWR_CTL	"SDH Power Control Register (SDH_PWR_CTL)" on page 27-55
0xFFC0 3904	SDH_CLK_CTL	"SDH Clock Control Register (SDH_CLK_CTL)" on page 27-55
0xFFC0 3908	SDH_ARGUMENT	"SDH Argument Register (SDH_ARGUMENT)" on page 27-57
0xFFC0 390C	SDH_COMMAND	"SDH Command Register (SDH_COMMAND)" on page 27-57
0xFFC0 3910	SDH_RESP_CMD	"SDH Response Command Register (SDH_RESP_CMD)" on page 27-58
0xFFC0 3914	SDH_RESPONSE0	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
0xFFC0 3918	SDH_RESPONSE1	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
0xFFC0 391C	SDH_RESPONSE2	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59
0xFFC0 3920	SDH_RESPONSE3	"SDH Response Registers (SDH_RESPONSEx)" on page 27-59

Table A-20. SDH Functional Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 3924	SDH_DATA_TIMER	"SDH Data Timer Register (SDH_DATA_TIMER)" on page 27-60
0xFFC0 3928	SDH_DATA_LGTH	"SDH Data Length Register (SDH_DATA_LGTH)" on page 27-61
0xFFC0 392C	SDH_DATA_CTL	"SDH Data Control Register (SDH_DATA_CTL)" on page 27-61
0xFFC0 3930	SDH_DATA_CNT	"SDH Data Counter Register (SDH_DATA_CNT)" on page 27-62
0xFFC0 3934	SDH_STATUS	"SDH Status Register (SDH_STATUS)" on page 27-63
0xFFC0 3938	SDH_STATUS_CLR	"SDH Status Clear Register (SDH_STATUS_CLR)" on page 27-65
0xFFC0 393C	SDH_MASK0	"SDH Interrupt Mask Registers (SDH_MASKx)" on page 27-66
0xFFC0 3940	SDH_MASK1	"SDH Interrupt Mask Registers (SDH_MASKx)" on page 27-66
0xFFC0 3944	Reserved	–
0xFFC0 3948	SDH_FIFO_CNT	"SDH FIFO Counter Register (SDH_FIFO_CNT)" on page 27-68
0xFFC0 394C ... 0xFFC0 397C	Reserved	–
0xFFC0 3980	SDH_FIFOx	"SDH Data FIFO Register (SDH_FIFO)" on page 27-69
0xFFC0 3984 ... 0xFFC0 3988	Reserved	–
0xFFC0 39C0	SDH_E_STATUS	"SDH Exception Status Register (SDH_E_STATUS)" on page 27-69
0xFFC0 39C4	SDH_E_MASK	"SDH Exception Mask Register (SDH_E_MASK)" on page 27-70

Table A-20. SDH Functional Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 39C8	SDH_CFG	"SDH Configuration Register (SDH_CFG)" on page 27-71
0xFFC0 39CC	SDH_RD_WAIT_EN	"SDH Read Wait Enable Register (SDH_RD_WAIT_EN)" on page 27-72
0xFFC0 39D0 ... 0xFFC0 39EC	SDH_PIDx	"SDH Identification Registers (SDH_PIDx)" on page 27-73

ATAPI Registers

The ATAPI interface's memory-mapped registers (MMRs) regulate its operation. Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table A-21 lists the ATAPI memory-mapped registers, starting at base address 0xFFC0 3800. Register addresses are given relative to the base address.

Table A-21. ATAPI Core Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3800	ATAPI_CONTROL	"ATAPI Control (ATAPI_CONTROL) Register" on page 21-49
0xFFC0 3804	ATAPI_STATUS	"ATAPI Status (ATAPI_STATUS) Register" on page 21-51
0xFFC0 3808	ATAPI_DEV_ADDR	"ATAPI Device Address (ATAPI_DEV_ADDR) Register" on page 21-52
0xFFC0 380C	ATAPI_DEV_TXBUF	"ATAPI Device Transmit Buffer (ATAPI_DEV_TXBUF) Register" on page 21-53

Table A-21. ATAPI Core Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 3810	ATAPI_DEV_RXBUF	"ATAPI Device Receive Buffer (ATAPI_DEV_RXBUF) Register" on page 21-54
0xFFC0 3814	ATAPI_INT_MASK	"ATAPI Interrupt Mask (ATAPI_INT_MASK) Register" on page 21-54
0xFFC0 3818	ATAPI_INT_STATUS	"ATAPI Interrupt Status (ATAPI_INT_STATUS) Register" on page 21-56
0xFFC0 381C	ATAPI_XFER_LEN	"ATAPI Transfer Length (ATAPI_XFER_LEN) Register" on page 21-58
0xFFC0 3820	ATAPI_LINE_STATUS	"ATAPI Line Status (ATAPI_LINE_STATUS) Register" on page 21-59
0xFFC0 3824	ATAPI_SM_STATE	"ATAPI State Machine Status (ATAPI_SM_STATE) Register" on page 21-59
0xFFC0 3828	ATAPI_TERMINATE	"ATAPI Host Terminate (ATAPI_TERMINATE) Register" on page 21-60
0xFFC0 382C	ATAPI_PIO_TFRCNT	"ATAPI PIO Transfer Count (ATAPI_PIO_TFRCNT) Register" on page 21-61
0xFFC0 3830	ATAPI_DMA_TFRCNT	"ATAPI Multiword DMA Transfer Count (ATAPI_MULTI_TFRCNT) Register" on page 21-61
0xFFC0 3834	ATAPI_ULTRA_IN_TFRCNT	"ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT) Register" on page 21-62

Table A-21. ATAPI Core Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 3838	ATAPI_ULTRA_OUT_TFRCNT	"ATAPI Ultra DMA OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT) Register" on page 21-63
0xFFC0 3840	ATAPI_REG_TIM_0	"ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0) Register" on page 21-63
0xFFC0 3844	ATAPI_PIO_TIM_0	"ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0) Register" on page 21-64
0xFFC0 3848	ATAPI_PIO_TIM_1	"ATAPI Programmed I/O Timing 1 (ATAPI_PIO_TIM_1) Register" on page 21-64
0xFFC0 3850	ATAPI_MULTI_TIM_0	"ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0) Register" on page 21-65
0xFFC0 3854	ATAPI_MULTI_TIM_1	"ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1) Register" on page 21-65
0xFFC0 3858	ATAPI_MULTI_TIM_2	"ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2) Register" on page 21-66
0xFFC0 3860	ATAPI_ULTRA_TIM_0	"ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0) Register" on page 21-66
0xFFC0 3864	ATAPI_ULTRA_TIM_1	"ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1) Register" on page 21-67
0xFFC0 3868	ATAPI_ULTRA_TIM_2	"ATAPI Ultra DMA Timing 2 (ATAPI_ULTRA_TIM_2) Register" on page 21-67
0xFFC0 386C	ATAPI_ULTRA_TIM_3	"ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3) Register" on page 21-68

USB OTG Registers

Descriptions and bit diagrams for each of these MMRs are provided in the following sections. See Table A-22.

Table A-22. USB OTG Memory-Mapped Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3C00	USB_FADDR	"USB Function Address (USB_FADDR) Register" on page 26-102
0xFFC0 3C04	USB_POWER	"USB Power Management (USB_POWER) Register" on page 26-99
0xFFC0 3C08	USB_INTRTX	"USB Transmit Interrupt (USB_INTRTX) Register" on page 26-105
0xFFC0 3C0C	USB_INTRRX	"USB Receive Interrupt (USB_INTRRX) Register" on page 26-106
0xFFC0 3C10	USB_INTRTXE	"USB Transmit Interrupt Enable (USB_INTRTXE) Register" on page 26-107
0xFFC0 3C14	USB_INTRRXE	"USB Receive Interrupt Enable (USB_INTRRXE) Register" on page 26-108
0xFFC0 3C18	USB_INTRUSB	"USB Common Interrupts (USB_INTRUSB) Register" on page 26-109
0xFFC0 3C1C	USB_INTRUSBE	"USB Common Interrupt Enable (USB_INTRUSBE) Register" on page 26-110
0xFFC0 3C20	USB_FRAME	"USB Frame Number (USB_FRAME) Register" on page 26-111
0xFFC0 3C24	USB_INDEX	"USB Index (USB_INDEX) Register" on page 26-111
0xFFC0 3C28	USB_TESTMODE	"USB Test Mode (USB_TESTMODE) Register" on page 26-103 *(for Analog Devices internal use only)*
0xFFC0 3C2C	USB_GLOBINTR	"USB Global Interrupt (USB_GLOBINTR) Register" on page 26-104
0xFFC0 3C30	USB_GLOBAL_CTL	"USB Global Control (USB_GLOBAL_CTL) Register" on page 26-97

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
USB Packet Control – Indexed Registers		
0xFFC0 3C40	USB_TX_MAX_PACKET	"USB TX Max Packet (USB_TX_MAX_PACKET) Register" on page 26-112
0xFFC0 3C44	USB_CSR0	"USB Control/Status EP0 (USB_CSR0) Register" on page 26-113
0xFFC0 3C44	USB_TXCSR	"USB TX Control/Status EPx (USB_TXCSR) Register" on page 26-117
0xFFC0 3C48	USB_RX_MAX_PACKET	"USB RX Max Packet (USB_RX_MAX_PACKET) Register" on page 26-122
0xFFC0 3C4C	USB_RXCSR	"USB RX Control/Status (USB_RXCSR) Register" on page 26-123
0xFFC0 3C50	USB_COUNT0	"USB Count 0 (USB_COUNT0) Register" on page 26-128
0xFFC0 3C50	USB_RXCOUNT	"USB RX Byte Count EPx (USB_RXCOUNT) Register" on page 26-128
0xFFC0 3C54	USB_TXTYPE	"USB TX Type (USB_TXTYPE) Register" on page 26-129
0xFFC0 3C58	USB_NAKLIMIT0	"USB NAK Limit 0 (USB_NAKLIMIT0) Register" on page 26-130
0xFFC0 3C58	USB_TXINTERVAL	"USB TX Interval (USB_TXINTERVAL) Register" on page 26-130
0xFFC0 3C5C	USB_RXTYPE	"USB RX Type (USB_RXTYPE) Register" on page 26-131
0xFFC0 3C60	USB_RXINTERVAL	"USB RX Interval (USB_RXINTERVAL) Register" on page 26-132
0xFFC0 3C68	USB_TXCOUNT	"USB TX Byte Count EPx (USB_TXCOUNT) Register" on page 26-133

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint FIFO Registers	
0xFFC0 3C80	USB_EP0_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3C88	USB_EP1_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3C90	USB_EP2_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3C98	USB_EP3_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3CA0	USB_EP4_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3CA8	USB_EP5_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3CB0	USB_EP6_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
0xFFC0 3CB8	USB_EP7_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-134
	USB OTG Control Registers	
0xFFC0 3D00	USB_OTG_DEV_CTL	"USB OTG Device Control (USB_OTG_DEV_CTL) Register" on page 26-134
0xFFC0 3D04	USB_OTG_VBUS_IRQ	"USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register" on page 26-136
0xFFC0 3D08	USB_OTG_VBUS_MASK	"USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register" on page 26-137

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB PHY Control Registers	
0xFFC0 3D48	USB_LINKINFO	"USB Link Info (USB_LINKINFO) Register" on page 26-138
0xFFC0 3D4C	USB_VPLEN	"USB VBUS Pulse Length (USB_VPLEN) Register" on page 26-139
0xFFC0 3D50	USB_HS_EOF1	"USB High-Speed EOF 1 (USB_HS_EOF1) Register" on page 26-139
0xFFC0 3D54	USB_FS_EOF1	"USB Full-Speed EOF 1 (USB_FS_EOF1) Register" on page 26-140
0xFFC0 3D58	USB_LS_EOF1	"USB Low-Speed EOF 1 (USB_LS_EOF1) Register" on page 26-140
0xFFC0 3DE0	USB_APHY_CNTRL	"USB APHY Control 2 (USB_APHY_CNTRL2) Register" on page 26-141 *(for Analog Devices internal use only)*
0xFFC0 3DE4	USB_APHY_CALIB	*(for Analog Devices internal use only)*
0xFFC0 3DE8	USB_APHY_CNTRL2	Used to prevent re-enumeration after the processor goes into hibernate mode. "USB APHY Control 2 (USB_APHY_CNTRL2) Register" on page 26-141
0xFFC0 3DEC	USB_PHY_TEST	*(for Analog Devices internal use only)*
0xFFC0 3DF0	USB_PLLOSC_CTRL	"USB PLL OSC Control (USB_PLLOSC_CTRL) Register" on page 26-142
0xFFC0 3DF4	USB_SRP_CLKDIV	"USB SRP Clock Divider (USB_SRP_CLKDIV) Register" on page 26-143

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 0 Control Registers	
0xFFC0 3E00	USB_EP_NI0_TXMAXP	Maximum packet size for host Tx endpoint0
0xFFC0 3E04	USB_EP_NI0_TXCSR	Control Status register for endpoint 0
0xFFC0 3E08	USB_EP_NI0_RXMAXP	Maximum packet size for host Rx endpoint0
0xFFC0 3E0C	USB_EP_NI0_RXCSR	Control Status register for host Rx endpoint0
0xFFC0 3E10	USB_EP_NI0_ RXCOUNT	Number of bytes received in endpoint 0 FIFO
0xFFC0 3E14	USB_EP_NI0_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint0
0xFFC0 3E18	USB_EP_NI0_TXINTER-VAL	Sets the NAK response timeout on endpoint 0
0xFFC0 3E1C	USB_EP_NI0_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint0
0xFFC0 3E20	USB_EP_NI0_RXINTER-VAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint0

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
USB Endpoint 1 Control Registers		
0xFFC0 3E28	USB_EP_NI0_ TXCOUNT	Number of bytes to be written to the endpoint0 Tx FIFO
0xFFC0 3E40	USB_EP_NI1_TXMAXP	Maximum packet size for host Tx endpoint1
0xFFC0 3E44	USB_EP_NI1_TXCSR	Control Status register for endpoint1
0xFFC0 3E48	USB_EP_NI1_RXMAXP	Maximum packet size for host Rx endpoint1
0xFFC0 3E4C	USB_EP_NI1_RXCSR	Control Status register for host Rx endpoint1
0xFFC0 3E50	USB_EP_NI1_ RXCOUNT	Number of bytes received in endpoint1 FIFO
0xFFC0 3E54	USB_EP_NI1_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint1
0xFFC0 3E58	USB_EP_NI1_TXINTER-VAL	Sets the NAK response timeout on endpoint1
0xFFC0 3E5C	USB_EP_NI1_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint1
0xFFC0 3E60	USB_EP_NI1_RXINTER-VAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint1

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 2 Control Registers	
0xFFC0 3E68	USB_EP_NI1_ TXCOUNT	Number of bytes to be written to the+H102 endpoint1 Tx FIFO
0xFFC0 3E80	USB_EP_NI2_TXMAXP	Maximum packet size for host Tx endpoint2
0xFFC0 3E84	USB_EP_NI2_TXCSR	Control Status register for endpoint2
0xFFC0 3E88	USB_EP_NI2_RXMAXP	Maximum packet size for host Rx endpoint2
0xFFC0 3E8C	USB_EP_NI2_RXCSR	Control Status register for host Rx endpoint2
0xFFC0 3E90	USB_EP_NI2_ RXCOUNT	Number of bytes received in endpoint2 FIFO
0xFFC0 3E94	USB_EP_NI2_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint2
0xFFC0 3E98	USB_EP_NI2_TXINTER-VAL	Sets the NAK response timeout on endpoint2
0xFFC0 3E9C	USB_EP_NI2_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint2
0xFFC0 3EA0	USB_EP_NI2_RXINTER-VAL	Sets the polling interval for interrupt and isochro-nous transfers or the NAK response timeout on bulk transfers for host Rx endpoint2

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 3 Control Registers	
0xFFC0 3EA8	USB_EP_NI2_ TXCOUNT	Number of bytes to be written to the endpoint2 Tx FIFO
0xFFC0 3EC0	USB_EP_NI3_TXMAXP	Maximum packet size for host Tx endpoint3
0xFFC0 3EC4	USB_EP_NI3_TXCSR	Control Status register for endpoint3
0xFFC0 3EC8	USB_EP_NI3_RXMAXP	Maximum packet size for host Rx endpoint3
0xFFC0 3ECC	USB_EP_NI3_RXCSR	Control Status register for host Rx endpoint3
0xFFC0 3ED0	USB_EP_NI3_ RXCOUNT	Number of bytes received in endpoint3 FIFO
0xFFC0 3ED4	USB_EP_NI3_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint3
0xFFC0 3ED8	USB_EP_NI3_TXINTER-VAL	Sets the NAK response timeout on endpoint3
0xFFC0 3EDC	USB_EP_NI3_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint3
0xFFC0 3EE0	USB_EP_NI3_RXINTER-VAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint3

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 4 Control Registers	
0xFFC0 3EE8	USB_EP_NI3_ TXCOUNT	Number of bytes to be written to the H124endpoint3 Tx FIFO
0xFFC0 3F00	USB_EP_NI4_TXMAXP	Maximum packet size for host Tx endpoint4
0xFFC0 3F04	USB_EP_NI4_TXCSR	Control Status register for endpoint4
0xFFC0 3F08	USB_EP_NI4_RXMAXP	Maximum packet size for host Rx endpoint4
0xFFC0 3F0C	USB_EP_NI4_RXCSR	Control Status register for host Rx endpoint4
0xFFC0 3F10	USB_EP_NI4_ RXCOUNT	Number of bytes received in endpoint4 FIFO
0xFFC0 3F14	USB_EP_NI4_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint4
0xFFC0 3F18	USB_EP_NI4_TXINTER-VAL	Sets the NAK response timeout on endpoint4
0xFFC0 3F1C	USB_EP_NI4_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint4
0xFFC0 3F20	USB_EP_NI4_RXINTER-VAL	Sets the polling interval for interrupt and isochro-nous transfers or the NAK response timeout on bulk transfers for host Rx endpoint4

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 5 Control Registers	
0xFFC0 3F28	USB_EP_NI4_ TXCOUNT	Number of bytes to be written to the endpoint4 Tx FIFO
0xFFC0 3F40	USB_EP_NI5_TXMAXP	Maximum packet size for host Tx endpoint5
0xFFC0 3F44	USB_EP_NI5_TXCSR	Control Status register for endpoint5
0xFFC0 3F48	USB_EP_NI5_RXMAXP	Maximum packet size for host Rx endpoint5
0xFFC0 3F4C	USB_EP_NI5_RXCSR	Control Status register for host Rx endpoint5
0xFFC0 3F50	USB_EP_NI5_ RXCOUNT	Number of bytes received in endpoint5 FIFO
0xFFC0 3F54	USB_EP_NI5_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint5
0xFFC0 3F58	USB_EP_NI5_TXINTER-VAL	Sets the NAK response timeout on endpoint5
0xFFC0 3F5C	USB_EP_NI5_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint5
0xFFC0 3F60	USB_EP_NI5_RXINTER-VAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint5

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 6 Control Registers	
0xFFC0 3F68	USB_EP_NI5_ TXCOUNT	Number of bytes to be written to the H145 endpoint5 Tx FIFO
0xFFC0 3F80	USB_EP_NI6_TXMAXP	Maximum packet size for host Tx endpoint6
0xFFC0 3F84	USB_EP_NI6_TXCSR	Control Status register for endpoint6
0xFFC0 3F88	USB_EP_NI6_RXMAXP	Maximum packet size for host Rx endpoint6
0xFFC0 3F8C	USB_EP_NI6_RXCSR	Control Status register for host Rx endpoint6
0xFFC0 3F90	USB_EP_NI6_ RXCOUNT	Number of bytes received in endpoint6 FIFO
0xFFC0 3F94	USB_EP_NI6_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint6
0xFFC0 3F98	USB_EP_NI6_TXINTER-VAL	Sets the NAK response timeout on endpoint6
0xFFC0 3F9C	USB_EP_NI6_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint6
0xFFC0 3FA0	USB_EP_NI6_RXINTER-VAL	Sets the polling interval for interrupt and isochro-nous transfers or the NAK response timeout on bulk transfers for host Rx endpoint6

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Endpoint 7 Control Registers	
0xFFC0 3FA8	USB_EP_NI6_ TXCOUNT	Number of bytes to be written to the endpoint6 Tx FIFO
0xFFC0 3FC0	USB_EP_NI7_TXMAXP	Maximum packet size for host Tx endpoint7
0xFFC0 3FC4	USB_EP_NI7_TXCSR	Control Status register for endpoint7
0xFFC0 3FC8	USB_EP_NI7_RXMAXP	Maximum packet size for host Rx endpoint7
0xFFC0 3FCC	USB_EP_NI7_RXCSR	Control Status register for host Rx endpoint7
0xFFC0 3FD0	USB_EP_NI7_ RXCOUNT	Number of bytes received in endpoint7 FIFO
0xFFC0 3FD4	USB_EP_NI7_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint7
0xFFC0 3FD8	USB_EP_NI7_TXINTER-VAL	Sets the NAK response timeout on endpoint7
0xFFC0 3FDC	USB_EP_NI7_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint7
0xFFC0 3FE0	USB_EP_NI7_RXINTER-VAL	Sets the polling interval for interrupt and isochro-nous transfers or the NAK response timeout on bulk transfers for host Rx endpoint7
0xFFC0 3FE8	USB_EP_NI7_ TXCOUNT	Number of bytes to be written to the endpoint7 Tx FIFO
	USB DMA Registers	
0xFFC0 4000	USB_DMA_INTERRUPT	"USB DMA Interrupt (USB_DMA_INTERRUPT) Register" on page 26-144

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Channel 0 Config Registers	
0xFFC0 4004	USB_DMA0CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 4008	USB_DMA0ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 400C	USB_DMA0ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 4010	USB_DMA0COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 4014	USB_DMA0COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148
	USB Channel 1 Config Registers	
0xFFC0 4024	USB_DMA1CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 4028	USB_DMA1ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 402C	USB_DMA1ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 4030	USB_DMA1COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 4034	USB_DMA1COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Channel 2 Config Registers	
0xFFC0 4044	USB_DMA2CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 4048	USB_DMA2ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 404C	USB_DMA2ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 4050	USB_DMA2COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 4054	USB_DMA2COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148
	USB Channel 3 Config Registers	
0xFFC0 4064	USB_DMA3CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 4068	USB_DMA3ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 406C	USB_DMA3ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 4070	USB_DMA3COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 4074	USB_DMA3COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Channel 4 Config Registers	
0xFFC0 4084	USB_DMA4CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 4088	USB_DMA4ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 408C	USB_DMA4ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 4090	USB_DMA4COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 4094	USB_DMA4COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148
	USB Channel 5 Config Registers	
0xFFC0 40A4	USB_DMA5CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 40A8	USB_DMA5ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 40AC	USB_DMA5ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 40B0	USB_DMA5COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 40B4	USB_DMA5COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148

Table A-22. USB OTG Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
	USB Channel 6 Config Registers	
0xFFC0 40C4	USB_DMA6CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 40C8	USB_DMA6ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 40CC	USB_DMA6ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 40D0	USB_DMA6COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 40D4	USB_DMA6COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148
	USB Channel 7 Config Registers	
0xFFC0 40E4	USB_DMA7CONTROL	"USB DMAx Control (USB_DMA_CONTROL) Registers" on page 26-144
0xFFC0 40E8	USB_DMA7ADDR LOW	"USB DMAx Address Low (USB_DMAxADDRLOW) Registers" on page 26-146
0xFFC0 40EC	USB_DMA7ADDR HIGH	"USB DMAx Address High (USB_DMAxADDRHIGH) Registers" on page 26-147
0xFFC0 40F0	USB_DMA7COUNT LOW	"USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers" on page 26-147
0xFFC0 40F4	USB_DMA7COUNT HIGH	"USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers" on page 26-148

External Bus Interface Unit Registers

External bus interface unit (EBIU) registers (0xFFC0 0A00 – 0xFFC0 0AFF) are listed in Table A-23.

Table A-23. EBIU Memory-Mapped Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0A00	EBIU_AMGCTL	"Asynchronous Memory Global Control Register (EBIU_AMGCTL)" on page 5-57
0xFFC0 0A04	EBIU_AMBCTL0	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 5-59
0xFFC0 0A08	EBIU_AMBCTL1	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 5-59
0xFFC0 0A0C	EBIU_MBSCTL	"Memory Bank Select Control Register (EBIU_MBSCTL)" on page 5-63
0xFFC0 0A10	EBIU_ARBSTAT	"EBIU Arbitration Status Register (EBIU_ARBSTAT)" on page 5-69
0xFFC0 0A14	EBIU_MODE	"Flash Memory Bank Control Registers (EBIU_FCTL, EBIU_MODE)" on page 5-64
0xFFC0 0A18	EBIU_FCTL	"Flash Memory Bank Control Registers (EBIU_FCTL, EBIU_MODE)" on page 5-64
0xFFC0 0A20	EBIU_DDRCTL0	"Memory Control Register 0 (EBIU_DDRCTL0)" on page 5-24)
0xFFC0 0A24	EBIU_DDRCTL1	"Memory Control Register 1 (EBIU_DDRCTL1)" on page 5-25
0xFFC0 0A28	EBIU_DDRCTL2	"Memory Control Register 2 (EBIU_DDRCTL2)" on page 5-26
0xFFC0 0A2C	EBIU_DDRCTL3	"Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices" on page 5-27 "Memory Control Register 3 (EBIU_DDRCTL3), Mobile DDR Devices" on page 5-28

Table A-23. EBIU Memory-Mapped Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0A30	EBIU_DDRQUE	"Queue Configuration Register (EBIU_DDRQUE)" on page 5-29
0xFFC0 0A34	EBIU_ERRADD	"Error Address Register (EBIU_ERRADD)" on page 5-32
0xFFC0 0A38	EBIU_ERRMST	"Error Master Register (EBIU_ERRMST)" on page 5-31
0xFFC0 0A3C	EBIU_RSTCTL	"Reset Control Register (EBIU_RSTCTL)" on page 5-30
0xFFC0 0A1C	Reserved	Reserved

DMA/Memory DMA Control Registers

DMA/Memory DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in Table A-24.

Table A-24. DMA/Memory DMA Control Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0B0C	DMA_TCPER	"DMA Traffic Control Counter Period (DMACx_TCPER) Registers" on page 7-119
0xFFC0 0B10	DMA_ TCCNT	"DMA Traffic Control Counter (DMACx_TCCNT) Registers" on page 7-119

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-25 and Table A-26. Table A-25 identifies the base address of each DMA channel, as well as the

register prefix that identifies the channel. Table A-26 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-25. 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_
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-26. DMA Register Suffix and Offset

Register Suffix	Offset From Base	Description
NEXT_DESC_PTR	0x00	"Next Descriptor Pointer (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR) Registers" on page 7-106
START_ADDR	0x04	"Start Address (DMAx_START_ADDR and MDMA_yy_START_ADDR) Registers" on page 7-88
CONFIG	0x08	"DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers" on page 7-79
X_COUNT	0x10	"Inner Loop Count (DMAx_X_COUNT and MDMA_yy_X_COUNT) Registers" on page 7-92
X_MODIFY	0x14	"Inner Loop Address Increment (DMAx_X_MODIFY and MDMA_yy_X_MODIFY) Registers" on page 7-97
Y_COUNT	0x18	"Outer Loop Count (DMAx_Y_COUNT and MDMA_yy_Y_COUNT) Registers" on page 7-99
Y_MODIFY	0x1C	"Outer Loop Address Increment (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY) Registers" on page 7-103
CURR_DESC_PTR	0x20	"Current Descriptor Pointer (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR) Registers" on page 7-108
CURR_ADDR	0x24	"Current Address (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR) Registers" on page 7-90
IRQ_STATUS	0x28	"Interrupt Status (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS) Registers" on page 7-84
PERIPHERAL_MAP	0x2C	"Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers" on page 7-77
CURR_X_COUNT	0x30	"Current Inner Loop Count (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT) Registers" on page 7-94
CURR_Y_COUNT	0x38	"Current Outer Loop Count (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT) Registers" on page 7-101

EPPI0 Registers

EPPI0 registers are listed in Table A-29.

Table A-27. EPPI0 Registers

Memory Mapped Address	Register Name	Description
0xFFC0 1000	EPPI0_STATUS	"EPPI Status (EPPIx_STATUS) Register" on page 15-77
0xFFC0 1004	EPPI0_HCOUNT	"EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT)" on page 15-93
0xFFC0 1008	EPPI0_HDELAY	"EPPI Horizontal Delay Register (EPPIx_HDELAY)" on page 15-92
0xFFC0 100C	EPPI0_VCOUNT	"EPPI Vertical Transfer Count Register (EPPIx_VCOUNT)" on page 15-91
0xFFC0 1010	EPPI0_VDELAY	"EPPI Vertical Delay Register (EPPIx_VDELAY)" on page 15-91
0xFFC0 1014	EPPI0_FRAME	"EPPI Lines per Frame Register (EPPIx_FRAME)" on page 15-90
0xFFC0 1018	EPPI0_LINE	"EPPI Samples per Line Register (EPPIx_LINE)" on page 15-90
0xFFC0 101C	EPPI0_CLKDIV	"EPPI Clock Divide Register (EPPIx_CLKDIV)" on page 15-93
0xFFC0 1020	EPPI0_CON-TROL	"EPPIx Control (EPPIx_CONTROL) Register" on page 15-80
0xFFC0 1024	EPPI0_FSIW_HBL	"EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL)" on page 15-94
0xFFC0 1028	EPPI0_FSIP_AVPL	"EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL)" on page 15-96
0xFFC0 102C	EPPI0_FS2W_LVB	"EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB)" on page 15-95

Table A-27. EPPI0 Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 1030	EPPI0_FS2P_LAVF	"EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF)" on page 15-97
0xFFC0 1034	EPPI0_CLIP	"EPPI Clipping Register (EPPIx_CLIP)" on page 15-98

EPPI1 Registers

EPPI1 registers are listed in Table A-29.

Table A-28. EPPI1 Registers

Memory Mapped Address	Register Name	Description
0xFFC0 1300	EPPI1_STATUS	"EPPI Status (EPPIx_STATUS) Register" on page 15-77
0xFFC0 1304	EPPI1_HCOUNT	"EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT)" on page 15-93
0xFFC0 1308	EPPI1_HDELAY	"EPPI Horizontal Delay Register (EPPIx_HDELAY)" on page 15-92
0xFFC0 130C	EPPI1_VCOUNT	"EPPI Vertical Transfer Count Register (EPPIx_VCOUNT)" on page 15-91
0xFFC0 1310	EPPI1_VDELAY	"EPPI Vertical Delay Register (EPPIx_VDELAY)" on page 15-91
0xFFC0 1314	EPPI1_FRAME	"EPPI Lines per Frame Register (EPPIx_FRAME)" on page 15-90
0xFFC0 1318	EPPI1_LINE	"EPPI Samples per Line Register (EPPIx_LINE)" on page 15-90
0xFFC0 131C	EPPI1_CLKDIV	"EPPI Clock Divide Register (EPPIx_CLKDIV)" on page 15-93
0xFFC0 1320	EPPI1_CONTROL	"EPPIx Control (EPPIx_CONTROL) Register" on page 15-80

Table A-28. EPPI1 Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 1324	EPPI1_FSIW_HBL	"EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL)" on page 15-94
0xFFC0 1328	EPPI1_FSIP_AVPL	"EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL)" on page 15-96
0xFFC0 132C	EPPI1_FS2W_LVB	"EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB)" on page 15-95
0xFFC0 1330	EPPI1_FS2P_LAVF	"EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF)" on page 15-97
0xFFC0 1334	EPPI1_CLIP	"EPPI Clipping Register (EPPIx_CLIP)" on page 15-98

EPPI2 Registers

EPPI2 registers are listed in Table A-29.

Table A-29. EPPI2 Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2900	EPPI2_STATUS	"EPPI Status (EPPIx_STATUS) Register" on page 15-77
0xFFC0 2904	EPPI2_HCOUNT	"EPPI Horizontal Transfer Count Register (EPPIx_HCOUNT)" on page 15-93
0xFFC0 2908	EPPI2_HDELAY	"EPPI Horizontal Delay Register (EPPIx_HDELAY)" on page 15-92
0xFFC0 290C	EPPI2_VCOUNT	"EPPI Vertical Transfer Count Register (EPPIx_VCOUNT)" on page 15-91
0xFFC0 2910	EPPI2_VDELAY	"EPPI Vertical Delay Register (EPPIx_VDELAY)" on page 15-91
0xFFC0 2914	EPPI2_FRAME	"EPPI Lines per Frame Register (EPPIx_FRAME)" on page 15-90

Table A-29. EPPI2 Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2918	EPPI2_LINE	"EPPI Samples per Line Register (EPPIx_LINE)" on page 15-90
0xFFC0 291C	EPPI2_CLKDIV	"EPPI Clock Divide Register (EPPIx_CLKDIV)" on page 15-93
0xFFC0 2920	EPPI2_CONTROL	"EPPIx Control (EPPIx_CONTROL) Register" on page 15-80
0xFFC0 2924	EPPI2_FSIW_HBL	"EPPI FS1 Width Register/EPPI Horizontal Blanking Samples per Line Register (EPPIx_FS1W_HBL)" on page 15-94
0xFFC0 2928	EPPI2_FSIP_AVPL	"EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (EPPIx_FS1P_AVPL)" on page 15-96
0xFFC0 292C	EPPI2_FS2W_LVB	"EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIx_FS2W_LVB)" on page 15-95
0xFFC0 2930	EPPI2_FS2P_LAVF	"EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (EPPIx_FS2P_LAVF)" on page 15-97
0xFFC0 2934	EPPI2_CLIP	"EPPI Clipping Register (EPPIx_CLIP)" on page 15-98

Host DMA Registers

Host DMA registers are listed in Table A-30.

Table A-30. Host DMA Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3A00	HOST_CONTROL	"HOSTDP Control Register" on page 8-25

Table A-30. Host DMA Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 3A04	HOST_STATUS	"HOSTDP Status Register" on page 8-27
0xFFC0 3A08	HOST_TIMEOUT	"HOSTDP Timeout Register" on page 8-29

PIXC Registers

The Pixel Compositor has memory-mapped registers (MMRs) that regulate its operation. These registers are listed in Table A-31. Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table A-31. PIXC Memory-Mapped Registers

Memory Mapped Address	Register Name	Description
0xFFC0 4400	PIXC_CTL	"PIXC Control (PIXC_CTL) Register" on page 28-37
0xFFC0 4404	PIXC_PPL	"PIXC Pixels Per Line (PIXC_PPL) Register" on page 28-38
0xFFC0 4408	PIXC_LPF	"PIXC Lines Per Frame (PIXC_LPF) Register" on page 28-38
0xFFC0 440C	PIXC_AHSTART	"PIXC Horizontal Start (PIXC_xHSTART) Registers" on page 28-39
0xFFC0 4410	PIXC_AHEND	"PIXC Horizontal End (PIXC_xHEND) Registers" on page 28-39
0xFFC0 4414	PIXC_AVSTART	"PIXC Vertical Start (PIXC_xVSTART) Registers" on page 28-40
0xFFC0 4418	PIXC_AVEND	"PIXC Vertical End (PIXC_xVEND) Registers" on page 28-40
0xFFC0 441C	PIXC_ATRANSP	"PIXC Transparency Value (PIXC_xTRANSP) Registers" on page 28-41

Table A-31. PIXC Memory-Mapped Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 4420	PIXC_BHSTART	"PIXC Horizontal Start (PIXC_xHSTART) Registers" on page 28-39
0xFFC0 4424	PIXC_BHEND	"PIXC Horizontal End (PIXC_xHEND) Registers" on page 28-39
0xFFC0 4428	PIXC_BVSTART	"PIXC Vertical Start (PIXC_xVSTART) Registers" on page 28-40
0xFFC0 442C	PIXC_BVEND	"PIXC Vertical End (PIXC_xVEND) Registers" on page 28-40
0xFFC0 4430	PIXC_BTRANSP	"PIXC Transparency Value (PIXC_xTRANSP) Registers" on page 28-41
0xFFC0 443C	PIXC_INTRSTAT	"PIXC Interrupt Status (PIXC_INTRSTAT) Register" on page 28-41
0xFFC0 4440	PIXC_RYCON	"PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register" on page 28-42
0xFFC0 4444	PIXC_GUCON	"PIXC G/U Conversion Coefficient (PIXC_GUCON) Register" on page 28-43
0xFFC0 4448	PIXC_BVCON	"PIXC B/V Conversion Coefficient (PIXC_BVCON) Register" on page 28-44
0xFFC0 444C	PIXC_CCBIAS	"PIXC Color Conversion Bias (PIXC_CCBIAS) Register" on page 28-45
0xFFC0 4450	PIXC_TC	"PIXC Transparency Color Value (PIXC_TC) Register" on page 28-46

Ports Registers

Table A-32 lists the registers for port control and Table A-33 lists the registers for pin interrupt programming.

Table A-32. Port Control Registers (Multiplexing and GPIO)

Memory Mapped Address	Register Name	Description
0xFFC014C0	PORTA_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC014C4	PORTA data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC014C8	PORTA_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC014CC	PORTA_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC014D0	PORTA_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC014D4	PORTA_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC014D8	PORTA_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC014DC	PORTA_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC014E0	PORTB_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC014E4	PORTB data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC014E8	PORTB_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC014EC	PORTB_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51

Table A-32. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC014F0	PORTB_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC014F4	PORTB_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC014F8	PORTB_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC014FC	PORTB_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC01500	PORTC_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC01504	PORTC data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01508	PORTC_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC0150C	PORTC_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01510	PORTC_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01514	PORTC_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01518	PORTC_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC0151C	PORTC_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC01520	PORTD_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC01524	PORTD data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01528	PORTD_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51

Table A-32. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0152C	PORTD_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01530	PORTD_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01534	PORTD_DIR_ CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01538	PORTD_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC0153C	PORTD_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC01540	PORTE_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC01544	PORTE data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01548	PORTE_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC0154C	PORTE_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01550	PORTE_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01554	PORTE_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01558	PORTE_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC0155C	PORTE_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC01560	PORTF_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC01564	PORTF data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51

Table A-32. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC01568	PORTF_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC0156C	PORTF_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01570	PORTF_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01574	PORTF_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01578	PORTF_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC0157C	PORTF_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC01580	PORTG_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC01584	PORTG_DIR_ CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01588	PORTG_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC0158C	PORTG_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC01590	PORTG_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01594	PORTG data	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC01598	PORTG_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC0159C	PORTG_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC015A0	PORTH_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45

Table A-32. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC015A4	PORTH data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015A8	PORTH_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015AC	PORTH_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015B0	PORTH_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC015B4	PORTH_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC015B8	PORTH_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC015BC	PORTH_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46
0xFFC015C0	PORTI_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC015C4	PORTI data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015C8	PORTI_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015CC	PORTI_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_CLEAR) Registers" on page 9-51
0xFFC015D0	PORTI_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC015D4	PORTI_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Registers" on page 9-48
0xFFC015D8	PORTI_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC015DC	PORTI_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46

Table A-32. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC015E0	PORTJ_FER	"Port x Function Enable (PORTx_FER) Registers" on page 9-45
0xFFC015E4	PORTJ data	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC015E8	PORTJ_SET	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC015EC	PORTJ_CLEAR	"Port x GPIO Data (PORTx/ PORTx_SET/PORTx_ CLEAR) Registers" on page 9-51
0xFFC015F0	PORTJ_DIR_SET	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC015F4	PORTJ_DIR_CLEAR	"Port x GPIO Direction Set (PORTx_DIR_ SET/CLEAR) Registers" on page 9-48
0xFFC015F8	PORTJ_INEN	"Port x GPIO Input Enable (PORTx_INEN) Registers" on page 9-50
0xFFC015FC	PORTJ_MUX	"Port Multiplexer Control (PORTx_MUX) Registers" on page 9-46

Table A-33 lists the registers for pin interrupt programming.

Table A-33. Pin Interrupt Registers

Memory Mapped Address	Register Name	Description
0xFFC01400	PINT0_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01404	PINT0_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01408	PINT0_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55

Table A-33. Pin Interrupt Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0140C	PINT0_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63
0xFFC01410	PINT0_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC01414	PINT0_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC01418	PINT0_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC0141C	PINT0_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC01420	PINT0_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60
0xFFC01424	PINT0_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
0xFFC01430	PINT1_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01434	PINT1_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01438	PINT1_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
0xFFC0143C	PINT1_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63
0xFFC01440	PINT1_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC01444	PINT1_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC01448	PINT1_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC0144C	PINT1_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61

Table A-33. Pin Interrupt Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC01450	PINT1_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60
0xFFC01454	PINT1_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
0xFFC01460	PINT2_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01464	PINT2_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01468	PINT2_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
0xFFC0146 C	PINT2_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Regis- ters" on page 9-63
0xFFC01470	PINT2_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_ CLEAR) Register Pairs" on page 9-58
0xFFC01474	PINT2_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_ CLEAR) Register Pairs" on page 9-58
0xFFC01478	PINT2_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC0147 C	PINT2_INVERT_ CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC01480	PINT2_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60
0xFFC01484	PINT2_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55
0xFFC01490	PINT3_MASK_SET	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01494	PINT3_MASK_CLEAR	"Pin Interrupt Mask (PINTx_MASK_SET/ PINTx_ MASK_CLEAR) Register Pairs" on page 9-54
0xFFC01498	PINT3_REQUEST	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55

Table A-33. Pin Interrupt Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0149C	PINT3_ASSIGN	"Pin Interrupt Assignment (PINTx_ASSIGN) Registers" on page 9-63
0xFFC014A0	PINT3_EDGE_SET	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC014A4	PINT3_EDGE_CLEAR	"Interrupt Edge (PINTx_EDGE_SET/ PINTx_EDGE_CLEAR) Register Pairs" on page 9-58
0xFFC014A8	PINT3_INVERT_SET	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC014AC	PINT3_INVERT_CLEAR	"Pin Interrupt Invert Set (PINTx_INVERT_SET/ PINTx_INVERT_CLEAR) Registers" on page 9-61
0xFFC014B0	PINT3_PINSTATE	"Pin Interrupt Pin State (PINTx_PINSTATE) Register" on page 9-60
0xFFC014B4	PINT3_LATCH	"Interrupt Request and Latch (PINTx_REQUEST/ PINTx_LATCH) Registers" on page 9-55

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in Table A-34.

Table A-34. Timer Registers

Memory Mapped Address	Register Name	Description
0xFFC0 1600	TIMER0_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1604	TIMER0_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1608	TIMER0_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47

Table A-34. Timer Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 160C	TIMER0_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1610	TIMER1_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1614	TIMER1_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1618	TIMER1_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 161C	TIMER1_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1620	TIMER2_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1624	TIMER2_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1628	TIMER2_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 162C	TIMER2_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1630	TIMER3_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1634	TIMER3_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1638	TIMER3_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 163C	TIMER3_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1640	TIMER4_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1644	TIMER4_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44

Table A-34. Timer Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 1648	TIMER4_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 164C	TIMER4_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1650	TIMER5_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1654	TIMER5_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1658	TIMER5_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 165C	TIMER5_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1660	TIMER6_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1664	TIMER6_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1668	TIMER6_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 166C	TIMER6_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1670	TIMER7_CONFIG	"Timer Configuration (TIMERx_CONFIG) Registers" on page 10-42
0xFFC0 1674	TIMER7_COUNTER	"Timer Counter (TIMERx_COUNTER) Registers" on page 10-44
0xFFC0 1678	TIMER7_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 167C	TIMER7_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 10-47
0xFFC0 1680	TIMER_ENABLE	"Timer Enable (TIMER_ENABLEx) Registers" on page 10-38

Table A-34. Timer Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 1684	TIMER_DISABLE	"Timer Disable (TIMER_DISABLEx) Registers" on page 10-39
0xFFC0 1688	TIMER_STATUS	"Timer Status (TIMER_STATUSx) Registers" on page 10-40

CANx Registers

CANx registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in Table A-35 through Table A-38.

Table A-35. CANx Control and Configuration Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2A00	CANx_MC1	"Mailbox Configuration (CANx_MCx) Registers" on page 31-69
0xFFC0 2A04	CANx_MD1	"Mailbox Direction (CANx_MDx) Registers" on page 31-70
0xFFC0 2A08	CANx_TRS1	"Transmission Request Set (CANx_TRSx) Registers" on page 31-74
0xFFC0 2A0C	CANx_TRR1	"Transmission Request Reset (CANx_TRRx) Registers" on page 31-75
0xFFC0 2A10	CANx_TA1	"Transmission Acknowledge (CANx_TAx) Registers" on page 31-77
0xFFC0 2A14	CANx_AA1	"Abort Acknowledge (CANx_AAx) Registers" on page 31-76
0xFFC0 2A18	CANx_RMP1	"Receive Message Pending (CANx_RMPx) Registers" on page 31-71
0xFFC0 2A1C	CANx_RML1	"Receive Message Lost (CANx_RMLx) Registers" on page 31-72

Table A-35. CANx Control and Configuration Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2A20	CANx_MBTIF1	"Mailbox Transmit Interrupt Flag (CANx_MBTIFx) Registers" on page 31-80
0xFFC0 2A24	CANx_MBRIF1	"Mailbox Receive Interrupt Flag (CANx_MBRIFx) Registers" on page 31-81
0xFFC0 2A28	CANx_MBIM1	"Mailbox Interrupt Mask (CANx_MBIMx) Registers" on page 31-79
0xFFC0 2A2C	CANx_RFH1	"Remote Frame Handling (CANx_RFHx) Registers" on page 31-78
0xFFC0 2A30	CANx_OPSS1	"Overwrite Protection/Single Shot Transmission (CANx_OPSSx) Register" on page 31-73
0xFFC0 2A40	CANx_MC2	"Mailbox Configuration (CANx_MCx) Registers" on page 31-69
0xFFC0 2A44	CANx_MD2	"Mailbox Direction (CANx_MDx) Registers" on page 31-70
0xFFC0 2A48	CANx_TRS2	"Transmission Request Set (CANx_TRSx) Registers" on page 31-74
0xFFC0 2A4C	CANx_TRR2	"Transmission Request Reset (CANx_TRRx) Registers" on page 31-75
0xFFC0 2A50	CANx_TA2	"Transmission Acknowledge (CANx_TAx) Registers" on page 31-77
0xFFC0 2A54	CANx_AA2	"Abort Acknowledge (CANx_AAx) Registers" on page 31-76
0xFFC0 2A58	CANx_RMP2	"Receive Message Pending (CANx_RMPx) Registers" on page 31-71
0xFFC0 2A5C	CANx_RML2	"Receive Message Lost (CANx_RMLx) Registers" on page 31-72
0xFFC0 2A60	CANx_MBTIF2	"Mailbox Transmit Interrupt Flag (CANx_MBTIFx) Registers" on page 31-80
0xFFC0 2A64	CANx_MBRIF2	"Mailbox Receive Interrupt Flag (CANx_MBRIFx) Registers" on page 31-81
0xFFC0 2A68	CANx_MBIM2	"Mailbox Interrupt Mask (CANx_MBIMx) Registers" on page 31-79

Table A-35. CANx Control and Configuration Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2A6C	CANx_RFH2	"Remote Frame Handling (CANx_RFHx) Registers" on page 31-78
0xFFC0 2A70	CANx_OPSS2	"Overwrite Protection/Single Shot Transmission (CANx_OPSSx) Register" on page 31-73
0xFFC0 2A80	CANx_CLOCK	"CAN Clock (CANx_CLOCK) Registers" on page 31-47
0xFFC0 2A84	CANx_TIMING	"CAN Timing (CANx_TIMING) Registers" on page 31-48
0xFFC0 2A88	CANx_DEBUG	"CAN Debug (CANx_DEBUG) Registers" on page 31-47
0xFFC0 2A8C	CANx_STATUS	"Global CAN Status (CANx_STATUS) Registers" on page 31-46
0xFFC0 2A90	CANx_CEC	"Error Counter (CANx_CEC) Register" on page 31-84
0xFFC0 2A94	CANx_GIS	"Global CAN Interrupt Status (CANx_GIS) Registers" on page 31-49
0xFFC0 2A98	CANx_GIM	"Global CAN Interrupt Mask (CANx_GIM) Registers" on page 31-49
0xFFC0 2A9C	CANx_GIF	"Global CAN Interrupt Flag (CANx_GIF) Registers" on page 31-50
0xFFC0 2AA0	CANx_CONTROL	"Master Control (CANx_CONTROL) Registers" on page 31-45
0xFFC0 2AA4	CANx_INTR	"CAN Interrupt (CANx_INTR) Registers" on page 31-48
0xFFC0 2AAC	CANx_MBTD	"Temporary Mailbox Disable (CANx_MBTD) Register" on page 31-78
0xFFC0 2AB0	CANx_EWR	"Error Counter Warning Level (CANx_EWR) Register" on page 31-85
0xFFC0 2AB4	CANx_ESR	"Error Status (CANx_ESR) Register" on page 31-85
0xFFC0 2AC4	CANx_UCCNT	"Universal Counter (CANx_UCCNT) Register" on page 31-84

Table A-35. CANx Control and Configuration Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2AC8	CANx_UCRC	"Universal Counter Reload/Capture (CANx_UCRC) Register" on page 31-84
0xFFC0 2ACC	CANx_UCCNF	"Universal Counter Configuration Mode (CANx_UCCNF) Register" on page 31-83

Table A-36. CANx Mailbox Acceptance Mask Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2B00	CANx_AM00L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B04	CANx_ AM00H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B08	CANx_AM01L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B0C	CANx_ AM01H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B10	CANx_AM02L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B14	CANx_ AM02H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B18	CANx_AM03L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B1C	CANx_ AM03H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B20	CANx_AM04L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B24	CANx_ AM04H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B28	CANx_AM05L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B2C	CANx_ AM05H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B30	CANx_AM06L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50

Table A-36. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2B34	CANx_ AM06H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B38	CANx_AM07L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B3C	CANx_ AM07H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B40	CANx_AM08L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B44	CANx_ AM08H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B48	CANx_AM09L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B4C	CANx_ AM09H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B50	CANx_AM10L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B54	CANx_ AM10H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B58	CANx_AM11L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B5C	CANx_ AM11H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B60	CANx_AM12L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B64	CANx_ AM12H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B68	CANx_AM13L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B6C	CANx_ AM13H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B70	CANx_AM14L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B74	CANx_ AM14H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B78	CANx_AM15L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50

Table A-36. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2B7C	CANx_ AM15H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B80	CANx_AM16L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B84	CANx_ AM16H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B88	CANx_AM17L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B8C	CANx_ AM17H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B90	CANx_AM18L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B94	CANx_ AM18H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B98	CANx_AM19L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2B9C	CANx_ AM19H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BA0	CANx_AM20L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BA4	CANx_ AM20H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BA8	CANx_AM21L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BAC	CANx_ AM21H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BB0	CANx_AM22L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BB4	CANx_ AM22H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BB8	CANx_AM23L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BBC	CANx_ AM23H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BC0	CANx_AM24L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50

Table A-36. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 2BC4	CANx_ AM24H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BC8	CANx_AM25L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BCC	CANx_ AM25H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BD0	CANx_AM26L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BD4	CANx_ AM26H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BD8	CANx_AM27L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BDC	CANx_ AM27H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BE0	CANx_AM28L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BE4	CANx_ AM28H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BE8	CANx_AM29L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BEC	CANx_ AM29H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BF0	CANx_AM30L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BF4	CANx_ AM30H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BF8	CANx_AM31L	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50
0xFFC0 2BFC	CANx_ AM31H	"Acceptance Mask (CANx_AMxx) Registers" on page 31-50

Since each CANx 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-37 and Table A-38. Table A-37 identifies the base address of each CANx mailbox, as well as

the register prefix that identifies mailbox. Table A-38 then lists the register suffix and provides its offset from the base address.

As an example, the CANx 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-37. CANx Mailbox Base Addresses

Mailbox Identifier	MMR Base Address	Register Prefix
0	0xFFC0 2C00	CANx_MB00_
1	0xFFC0 2C20	CANx_MB01_
2	0xFFC0 2C40	CANx_MB02_
3	0xFFC0 2C60	CANx_MB03_
4	0xFFC0 2C80	CANx_MB04_
5	0xFFC0 2CA0	CANx_MB05_
6	0xFFC0 2CC0	CANx_MB06_
7	0xFFC0 2CE0	CANx_MB07_
8	0xFFC0 2D00	CANx_MB08_
9	0xFFC0 2D20	CANx_MB09_
10	0xFFC0 2D40	CANx_MB10_
11	0xFFC0 2D60	CANx_MB11_
12	0xFFC0 2D80	CANx_MB12_
13	0xFFC0 2DA0	CANx_MB13_
14	0xFFC0 2DC0	CANx_MB14_
15	0xFFC0 2DE0	CANx_MB15_

Table A-37. CANx Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
16	0xFFC0 2E00	CANx_MB16_
17	0xFFC0 2E20	CANx_MB17_
18	0xFFC0 2E40	CANx_MB18_
19	0xFFC0 2E60	CANx_MB19_
20	0xFFC0 2E80	CANx_MB20_
21	0xFFC0 2EA0	CANx_MB21_
22	0xFFC0 2EC0	CANx_MB22_
23	0xFFC0 2EE0	CANx_MB23_
24	0xFFC0 2F00	CANx_MB24_
25	0xFFC0 2F20	CANx_MB25_
26	0xFFC0 2F40	CANx_MB26_
27	0xFFC0 2F60	CANx_MB27_
28	0xFFC0 2F80	CANx_MB28_
29	0xFFC0 2FA0	CANx_MB29_
30	0xFFC0 2FC0	CANx_MB30_
31	0xFFC0 2FE0	CANx_MB31_

Table A-38. CANx Mailbox Register Suffix and Offset

Register Suffix	Offset From Base	Description
DATA0	0x00	"Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers" on page 31-61
DATA1	0x04	"Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers" on page 31-61

Table A-38. CANx Mailbox Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	Description
DATA2	0x08	"Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers" on page 31-61
DATA3	0x0C	"Mailbox Word 3–0 (CANx_MBxx_DATA3–0) Registers" on page 31-61
LENGTH	0x10	"Mailbox Word 4 (CANx_MBxx_LENGTH) Registers" on page 31-59
TIME-STAMP	0x14	"Mailbox Word 5 (CANx_MBxx_TIMESTAMP) Registers" on page 31-58
ID0	0x18	"Mailbox Word 6 (CANx_MBxx_ID0) Registers" on page 31-56
ID1	0x1C	"Mailbox Word 7 (CANx_MBxx_ID1) Registers" on page 31-54

Handshake MDMA Control Registers

Handshake MDMA registers (0xFFC0 3300 – 0xFFC0 33FF) are listed in Table A-39.

Table A-39. HMDMA Registers

Memory Mapped Address	Register Name	Description
0xFFC0 4500	HMDMA0_CONTROL	"Handshake MDMA Control (HMDMAx_CONTROL) Registers" on page 7-111
0xFFC0 4504	HMDMA0_ECINIT	"Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers" on page 7-117
0xFFC0 4508	HMDMA0_BCINIT	"Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers" on page 7-114
0xFFC0 450C	HMDMA0_ECURGENT	"Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers" on page 7-117

Table A-39. HMDMA Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 4510	HMDMA0_ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers" on page 7-118
0xFFC0 4514	HMDMA0_ECOUNT	"Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers" on page 7-116
0xFFC0 4518	HMDMA0_BCOUNT	"Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers" on page 7-115
0xFFC0 4540	HMDMA1_CONTROL	"Handshake MDMA Control (HMDMAx_CONTROL) Registers" on page 7-111
0xFFC0 4544	HMDMA1_ECINIT	"Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers" on page 7-117
0xFFC0 4548	HMDMA1_BCINIT	"Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers" on page 7-114
0xFFC0 454C	HMDMA1_ECURGENT	"Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers" on page 7-117
0xFFC0 4550	HMDMA1_ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers" on page 7-118
0xFFC0 4554	HMDMA1_ECOUNT	"Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers" on page 7-116
0xFFC0 4558	HMDMA1_BCOUNT	"Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers" on page 7-115

NAND Flash Controller Registers

Table A-28 lists all of the NFC memory-mapped registers.

Table A-40. NFC Memory-Mapped Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3B00	NFC_CTL	"NFC Control Register (NFC_CTL)" on page 20-20
0xFFC0 3B04	NFC_STAT	"NFC Status Register (NFC_STAT)" on page 20-21
0xFFC0 3B08	NFC_IRQSTAT	"NFC Interrupt Status Register (NFC_IRQSTAT)" on page 20-22
0xFFC0 3B0C	NFC_IRQMASK	"NFC Interrupt Mask Register (NFC_IRQMASK)" on page 20-23
0xFFC0 3B10	NFC_ECC0	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B14	NFC_ECC1	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B18	NFC_ECC2	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B1C	NFC_ECC3	"NFC ECC Registers (NFC_ECCx)" on page 20-23
0xFFC0 3B20	NFC_COUNT	"NFC Count Register (NFC_COUNT)" on page 20-25
0xFFC0 3B24	NFC_RST	"NFC Reset Register (NFC_RST)" on page 20-25
0xFFC0 3B28	NFC_PGCTL	"NFC Page Control Register (NFC_PGCTL)" on page 20-26
0xFCC0 3B2C	NFC_READ	"NFC Read Data Register (NFC_READ)" on page 20-26
0xFFC0 3B40	NFC_ADDR	"NFC Address Register (NFC_ADDR)" on page 20-27
0xFFC0 3B44	NFC_CMD	"NFC Command Register (NFC_CMD)" on page 20-28
0xFFC0 3B48	NFC_DATA_WR	"NFC Data Write Register (NFC_DATA_WR)" on page 20-29
0xFFC0 3B4C	NFC_DATA_RD	"NFC Data Read Register (NFC_DATA_RD)" on page 20-29

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in Table A-41.

Table A-41. Core Timer Registers

Memory Mapped Address	Register Name	Description
0xFFE0 3000	TCNTL	"Core Timer Control (TCNTL) Register" on page 11-5
0xFFE0 3004	TPERIOD	"Core Timer Period (TPERIOD) Register" on page 11-6
0xFFE0 3008	TSCALE	"Core Timer Scale (TSCALE) Register" on page 11-7
0xFFE0 300C	TCOUNT	"Core Timer Count (TCOUNT) Register" on page 11-5

Rotary Counter Registers

The rotary encoder interface has eight memory-mapped registers (MMRs) that regulate its operation. Refer to Table A-42 for an overview of all MMRs associated with the rotary encoder interface.

Table A-42. Rotary Counter Module Registers

Memory Mapped Address	Register Name	Description
0xFFC0 4200	CNT_CONFIG	"Configuration (CNT_CONFIG) Register" on page 13-26
0xFFC0 4204	CNT_IMASK	"Interrupt Mask (CNT_IMASK) Register" on page 13-28
0xFFC0 4208	CNT_STATUS	"Status (CNT_STATUS) Register" on page 13-28
0xFFC0 420C	CNT_COMMAND	"Command (CNT_COMMAND) Register" on page 13-29

Table A-42. Rotary Counter Module Registers (Cont'd)

Memory Mapped Address	Register Name	Description
0xFFC0 4210	CNT_ DEBOUNCE	"Debounce Prescale (CNT_DEBOUNCE) Register" on page 13-30
0xFFC0 4214	CNT_COUNTER	"Counter (CNT_COUNTER) Register" on page 13-31
0xFFC0 4218	CNT_MAX	"Boundary (CNT_MIN and CNT_MAX) Registers" on page 13-32
0xFFC0 421C	CNT_MIN	"Boundary (CNT_MIN and CNT_MAX) Registers" on page 13-32

Security Registers

There are three registers which provide information that can be used during security mode control and to return status of the Secure State Machine states (See Table A-43). These registers require privileged access depending on the operating state of the processor.

Table A-43. Security Registers

Memory Mapped Address	Register Name	Description
0xFFC0 4320	SECURE_SYSSWT	"Secured System Switches (SECURE_SYSSWT) Register" on page 16-57
0xFFC0 4324	SECURE_CON-TROL	"Secure Control (SECURE_CONTROL) Register" on page 16-64
0xFFC0 4328	SECURE_STATUS	"Secure Status (SECURE_STATUS) Register" on page 16-67

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in Table A-44.

Table A-44. Processor-Specific Memory Registers

Memory Mapped Address	Register Name	Description
0xFFE0 0004	DMEM_CONTROL	"Data Memory Control Register (DMEM_CONTROL)" on page 3-27
0xFFE0 0300	DTEST_COM-MAND	"Data Test Command Register (DTEST_COMMAND)" on page 3-44

B TEST FEATURES

This chapter discusses the test features of the processor and includes the following sections:

- "JTAG Standard" on page B-1
- "Boundary-Scan Architecture" on page B-2

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.

Private ADI JTAG emulation functionality has some modified behavior dependent on the access privileges associated with the state of the Secure State Machine operating mode. This is to ensure that sensitive information and processing performed within Secure Entry Mode and Secure Mode will not be compromised through JTAG. For more information about private ADI JTAG emulation functionality when security features are used, see Chapter 16, “Security”.

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.

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 $\overline{\text{TRST}}$ 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.

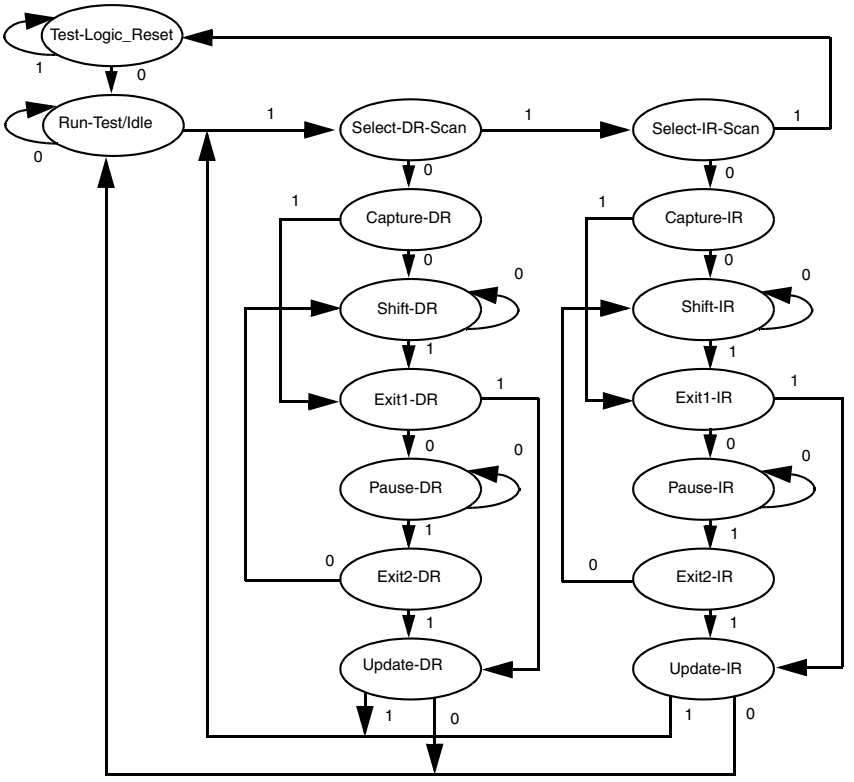


Figure B-1. TAP Controller State Diagram

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
IDCODE	00010	Device Identification

Figure B-2 shows the instruction bit scan ordering for the paths shown in Table B-2.

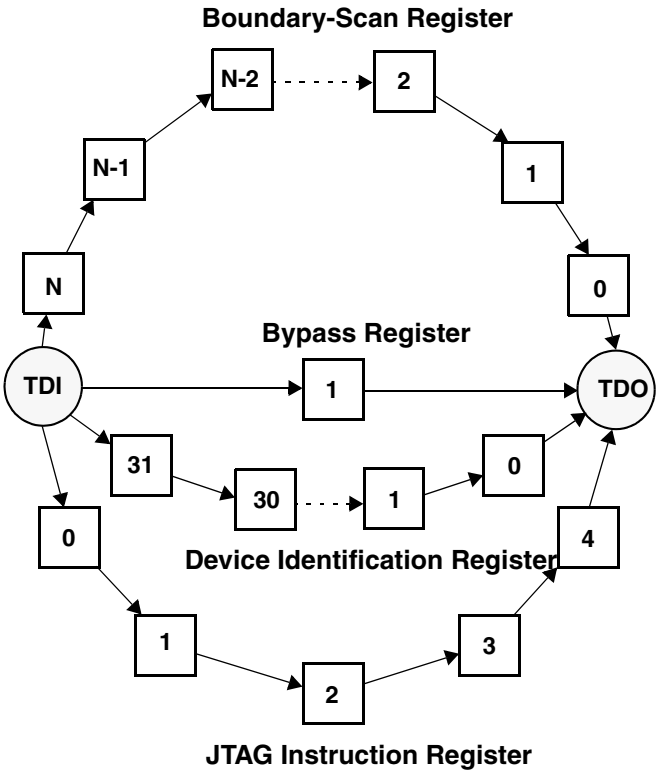


Figure B-2. Serial Scan Paths

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 overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

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.

IDCODE – Binary Code 00010

The IDCODE instruction selects the device identification register to be connected to TDI and TDO. This register allows identification of the device through the JTAG TAP.

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 are testable.

Digital Signal Processor (DSP).

An integrated circuit designated for high-speed manipulation of analog information that is converted into digital form.

direct branches.

Jump or call/return instructions that use absolute addresses that do not change at runtime (such as a program label), or they use a PC-relative address.

direct-mapped.

Cache architecture where each line has only one place that it can appear in the cache. Also described as 1-way associative.

Direct Memory Access (DMA).

A way of moving data between system devices and memory in which the data is transferred through a DMA port without involving the processor.

dirty, modified.

A state bit, stored along with the tag, indicating whether the data in the data cache line is changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

DMA.

See *Direct Memory Access*

DMA Access Bus (DAB).

A bus that provides a means for DMA channels to be accessed by the peripherals.

DMA chaining.

The linking or chaining of multiple direct memory access (DMA) sequences. In chained DMA, the I/O processor loads the next DMA descriptor into the DMA parameter registers when the current DMA finishes and autoinitializes the next DMA sequence.

DMA Core Bus (DCB).

A bus that provides a means for DMA channels to gain access to on-chip memory.

DMA descriptor registers.

Registers that hold the initialization information for a direct memory access (DMA) process.

DMA External Bus (DEB).

A bus that provides a means for DMA channels to gain access to off-chip memory.

DPMC (Dynamic Power Management Controller).

A processor's control block that allows the user to dynamically control the processor's performance characteristics and power dissipation.

DQM Data I/O Mask Function.

The SDQM[1:0] pins provide a byte-masking capability on 8-bit writes to SDRAM.

DRAM (Dynamic Random Access Memory).

A type of semiconductor memory in which the data is stored as electrical charges in an array of cells, each consisting of a capacitor and a transistor. The cells are arranged on a chip in a grid of rows and columns. Since the capacitors discharge gradually—and the cells lose their information—the array of cells has to be refreshed periodically.

DSP.

See *Digital Signal Processor*

EAB.

See *External Access Bus*

EBC.

See *External Bus Controller*

EBIU.

See *External Bus Interface Unit*

edge-sensitive interrupt.

A signal or interrupt the processor detects if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of CLKIN.

Endian format.

The ordering of bytes in a multibyte number.

EPB.

See *External Port Bus*

EPROM (Erasable Programmable Read-Only Memory).

A type of semiconductor memory in which the data is stored as electrical charges in isolated ("floating") transistor gates that retain their charges almost indefinitely without an external power supply. An EPROM is programmed by "injecting" charge into the floating gates—a process that requires relatively high voltage (usually 12V – 25V). Ultraviolet light, applied to the chip's surface through a quartz window in the package, discharges the floating gates, allowing the chip to be reprogrammed.

EVT (Event Vector Table).

A table stored in memory that contains sixteen 32-bit entries; each entry contains a vector address for an interrupt service routine (ISR). When an event occurs, instruction fetch starts at the address location in the corresponding EVT entry. See *ISR*.

exclusive, clean.

The state of a data cache line indicating the line is valid and the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

External Access Bus (EAB).

A bus mastered by the core memory management unit to access external memory.

External Bus Controller (EBC).

A component that provides arbitration between the External Access Bus (EAB) and the DMA External Bus (DEB), granting at most one requester per cycle.

External Bus Interface Unit (EBIU).

A component that provides glueless interfaces to external memories. It services requests for external memory from the core or from a DMA channel.

external port.

A channel or port that extends the processor's internal address and data buses off-chip, providing the processor's interface to off-chip memory and peripherals.

External Port Bus (EPB).

A bus that connects the output of the EBIU to external devices.

FFT (Fast Fourier Transform).

An algorithm for computing the Fourier transform of a set of discrete data values. The FFT expresses a finite set of data points, for example a periodic sampling of a real-world signal, in terms of its component frequencies. Or conversely, the FFT reconstructs a signal from the frequency data. The FFT can also be used to multiply two polynomials.

FIFO (First In, First Out).

A hardware buffer or data structure from which items are taken out in the same order they were put in.

flash memory.

A type of single transistor cell, erasable memory in which erasing can only be done in blocks or for the entire chip.

fully associative.

Cache architecture where each line can be placed anywhere in the cache.

glueless.

No external hardware is required.

Harvard architecture.

A processor memory architecture that uses separate buses for program and data storage. The two buses let the processor fetch a data word and an instruction word simultaneously.

HLL (High Level Language).

A programming language that provides some level of abstraction above assembly language, often using English-like statements, where each command or statement corresponds to several machine instructions.

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 through the phase locked loop (PLL) module.

internal memory bank.

There are up to 4 internal memory banks on a given SDRAM. Each of these banks can be accessed with the bank select lines `BA[1:0]`. The bank address can be thought of as part of the row address.

interrupt.

An event that suspends normal processing and temporarily diverts the flow of control through an interrupt service routine (ISR). See *ISR*.

invalid.

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

IrDA (Infrared Data Association).

A nonprofit trade association that established standards for ensuring the quality and interoperability of devices using the infrared spectrum.

isochronous.

Processes where data must be delivered within certain time constraints.

ISR (Interrupt Service Routine).

Software that is executed when a specific interrupt occurs. A table stored in low memory contains pointers, also called vectors, that direct the processor to the corresponding ISR. See *EVT*.

JTAG (Joint Test Action Group).

An IEEE Standards working group that defines the IEEE 1149.1 standard for a test access port for testing electronic devices.

JTAG port.

A channel or port that supports the IEEE standard 1149.1 JTAG standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

jump.

A permanent transfer of the program flow to another part of program memory.

latency.

The overhead time used to find the correct place for memory access and preparing to access it.

Least Recently Used algorithm.

Replacement algorithm used by cache that first replaces lines that have been unused for the longest time.

Least Significant Bit (LSB).

The last or rightmost bit in the normal representation of a binary number—the bit of a binary number giving the number of ones.

Length registers.

A Data Address Generator (DAG) register that specifies the range of addresses in a circular buffer.

Level 1 (L1) memory.

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

Level 2 (L2) memory.

Memory that is at least one level removed from the core. L2 memory has a larger capacity than L1 memory, but it requires additional latency to access.

level-sensitive interrupts.

A signal or interrupt that the processor detects if the input signal is low (active) when sampled on the rising edge of `CLKIN`.

LIFO (Last In, First Out).

A data structure from which the next item taken out is the most recent item put in.

little endian.

The native data store format of the processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte at the highest byte address of the data storage location.

loop.

A sequence of instructions that executes several times.

LRU.

See *Least Recently Used algorithm*

LSB.

See *Least Significant Bit*

MAC (Media Access Control).

The Ethernet MAC provides a 10/100Mbit/s Ethernet interface, compliant to IEEE Std. 802.3-2002, between an MII (Media Independent Interface) and the Blackfin peripheral subsystem.

MAC (Multiply/Accumulate).

A mathematical operation that multiplies two numbers and then adds a third to get the result (see *Multiply Accumulator*).

Memory Management Unit (MMU).

A component of the processor that supports protection and selective caching of memory by using cacheability protection lookaside buffers (CPLBs).

Mode register.

Internal configuration registers within SDRAM devices which allow specification of the SDRAM device's functionality.

modified addressing.

The process whereby the Data Address Generator (DAG) produces an address that is incremented by a value or the contents of a register.

Modify register.

A Data Address Generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

MMR (Memory-Mapped Register).

A specific location in main memory used by the processor as if it were a register.

MMU.

See *Memory Management Unit*

MSB (Most Significant Bit).

The first or leftmost bit in the normal representation of a binary number—the bit of a binary number with the greatest weight (2(n-1)).

multifunction computations.

The parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the computational units and memory accesses. The multiple operations perform the same as if they were in corresponding single function computations.

multiplier.

A computational unit that performs fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

NMI (Nonmaskable Interrupt).

A high priority interrupt that cannot be disabled by another interrupt.

NRZ (Non-return-to-Zero).

A binary encoding scheme in which a 1 is represented by a change in the signal and a 0 by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

NRZI (Non-return-to-Zero Inverted).

A binary encoding scheme in which a 0 is represented by a change in the signal and a 1 is represented by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

orthogonal.

The characteristic of being independent. An orthogonal instruction set allows any register to be used in an instruction that references a register.

PAB.

See *Peripheral Access Bus*

page size.

The amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row.

Parallel Peripheral Interface (PPI).

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins.

PC (Program Counter).

A register that contains the address of the next instruction to be executed.

peripheral.

Functional blocks not included as part of the core, and typically used to support system level operations.

Peripheral Access Bus (PAB).

A bus used to provide access to EBIU memory-mapped registers.

PF (Programmable Flag).

General-purpose I/O pins. Each PF pin can be individually configured as either an input or an output pin, and each PF pin can be further configured to generate an interrupt.

Phase-Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

PLL.

See *Phase-Locked Loop*

PPI.

See *Parallel Peripheral Interface*

precision.

The number of bits after the binary point in the storage format for the number.

post-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments after the instruction is executed.

precharge command.

The precharge command closes a specific active page in an internal bank and the precharge all command closes all 4 active pages in all 4 banks.

pre-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments before the instruction is executed.

PWM (Pulse Width Modulation).

Also called Pulse Duration Modulation (PDM), PWM is a pulse modulation technique in which the duration of the pulses is varied by the modulating voltage.

RAS (Row Address Strobe).

A signal sent from the SDC to a DRAM device to indicate validity of row address lines.

Real-Time Clock (RTC).

A component that generates timing pulses for the digital watch features of the processor, including time of day, alarm, and stopwatch countdown features.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

RTC.

See *Real-Time Clock*

RZ (Return-to-Zero modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 0 is represented by a change from the low voltage level to the high voltage level; a 1 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

RZI (Return-to-Zero-Inverted modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 1 is represented by a change from the low voltage level to the high voltage level; a 0 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

saturation (ALU saturation mode).

A state in which all positive fixed-point overflows return the maximum positive fixed-point number, and all negative overflows return the maximum negative number.

SDC (SDRAM Controller).

A configurable memory controller supporting a bank of synchronous memory consisting of SDRAM.

SDRAM (Synchronous Dynamic Random Access Memory).

A form of DRAM that includes a clock signal with its other control signals. This clock signal allows SDRAM devices to support "burst" access modes that clock out a series of successive bits.

SDRAM bank.

Region of external memory that can be configured to be 16M bytes, 32M bytes, 64M bytes, or 128M bytes and is selected by the $\overline{\text{SMS}}$ pin.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

serial ports (SPORTs).

A high speed synchronous input/output device on the processor. The processor uses two synchronous serial ports that provide inexpensive interfaces to a wide variety of digital and mixed-signal peripheral devices.

set.

A group of *N*-line storage locations in the ways of an *N*-way cache, selected by the index field of the address.

set associative.

Cache architecture that limits line placement to a number of sets (or ways).

shifter.

A computational unit that completes logical and arithmetic shifts.

SIC (System Interrupt Controller).

Part of the processor's two-level event control mechanism. The SIC works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core.

SIMD (Single Instruction, Multiple Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

SP (Stack Pointer).

A register that points to the top of the stack.

SPI.

See *Serial Peripheral Interface*

SRAM.

See *Static Random Access Memory*

stack.

A data structure for storing items that are to be accessed in Last In, First Out (LIFO) order. When a data item is added to the stack, it is "pushed"; when a data item is removed from the stack, it is "popped."

Static Random Access Memory (SRAM).

Very fast read/write memory that does not require periodic refreshing.

system.

The system includes the peripheral set (timers, real-time clock, programmable flags, UART, SPORTs, PPI, and SPIs), the external memory controller (EBIU), the memory DMA controller, as well as the interfaces between these peripherals, and the optional, external (off-chip) resources.

System clock (SCLK).

A component that delivers clock pulses at a frequency determined by a programmable divider ratio within the PLL.

System Interrupt Controller (SIC).

Component that maps and routes events from peripheral interrupt sources to the prioritized, general-purpose interrupt inputs of the Core Event Controller (CEC).

TAP (Test Access Port).

See *JTAG port*

TDM.

See *Time Division Multiplexing*

Time Division Multiplexing (TDM).

A method used for transmitting separate signals over a single channel. Transmission time is broken into segments, each of which carries one element. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of the 24 channels.

TWI.

See *Two-Wire Interface*

Two-Wire Interface (TWI).

The TWI controller allows a device to interface to an Inter IC bus as specified by the *Philips 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

'A' or 'B' device indicator (B_DEVICE) bit, 26-134, 26-136
'B' or 'A' device indicator (B_DEVICE) bit, 26-134, 26-136

Numerics

BCODE, 16-58, 16-59
OTP_EBIU_DDRCTL0, 17-118
16-bit flash interface, 19-8
16-bit SRAM, interface, 19-8
2D DMA, 7-19
BCODE, 17-106
BMODE, 16-58, 16-59, 17-106
ILOC, 3-9
OTP_EBIU_DDRCTL0, 17-118
OTP_EBIU_DDRCTL1, 17-118
8/16 bit mode (DATA_SIZE), 8-25
8-bit flash interface, 19-8
8-bit SRAM interface, 19-8

A

A11 (A11 element/coefficient) bits, 28-42
A11 element/coefficients (A11) bits, 28-42
A12 (A12 element/coefficient) bits, 28-42
A13 (A13 element/coefficient) bits, 28-42
A14 (A14 bias vector) bits, 28-45
A14 bias vector (A14) bits, 28-45
A21 (A21 element/coefficient) bits, 28-43
A21 element/coefficients (A21) bits, 28-43
A22 (A22 element/coefficient) bits, 28-43
A22 element/coefficients (A22) bits, 28-43
A23 (A23 element/coefficient) bits, 28-43
A23 element/coefficients (A23) bits, 28-43
A24 (A24 bias vector) bits, 28-45
A24 bias vector (A24) bits, 28-45
A31 (A31 element/coefficient) bits, 28-44
A31 element/coefficients (A31) bits, 28-44
A32 (A32 element/coefficient) bits, 28-44
A32 element/coefficients (A32) bits, 28-44
A33 (A33 element/coefficient) bits, 28-44
A33 element/coefficients (A33) bits, 28-44
A34 (A34 bias vector) bits, 28-45
A34 bias vector (A34) bits, 28-45
AAIF bit, 31-26, 31-50
AAIM bit, 31-26, 31-49
AAIS bit, 31-26, 31-49
AAn bit, 31-76
ABO bit, 31-45
abort acknowledge interrupt, CAN, 31-26
aborts, DMA, 7-37
acceptance mask filtering, CAN, 31-17
acceptance mask register (CAN_AMxxH), 31-51
acceptance mask register (CAN_AMxxL), 31-52
Access
  request priority, *See also* Arbitration
  DMA stalls to L1 or L2 memory, 2-13
  latency and throughput for L2 memory, 2-13
  priority, L2 port, 2-12

access denied interrupt, CAN, 31-26
access to unimplemented address interrupt, CAN, 31-27
ACKE bit, 31-85
active descriptor queue, and DMA synchronization, 7-70
active low/high frame syncs, serial port, 24-35
active mode, 1-31, 18-9
Active Mode (ACTIVE) bit, 29-14
ACTIVE_PLLDISABLED bit, 18-27
ACTIVE_PLLENABLED bit, 18-27
ACTS bit, 25-32
ADCs, connecting to, 24-2
address-tag compare operation, 3-17
ADIF bit, 31-26, 31-50
ADIM bit, 31-26, 31-49
ADIS bit, 31-26, 31-49
advanced technology attachment packet interface, 21-1
A_HEND (overlay A horizontal end) bits, 28-39
A_HSTART (overlay A horizontal start) bits, 28-39
alarm clock, RTC, 14-2
alarm interrupt enable bit, 14-21
A-law companding, 24-26, 24-31
alignment exceptions, 3-76
All Bypass-MXVR Disabled Mode, 29-13
Allocation Table, 29-56
Allocation Table Updated (ATU) interrupt event, 29-33
Allocation Table Updated interrupt enable, 29-45
alternate frame sync mode, 24-38
alternate timing, serial port, 24-38
AMC, 1-14
AME bit, 31-54
AMIDE bit, 31-51
AMS, 5-54
ANAK bit, 23-36, 23-38
AP Data field, 29-136
AP Destination Address, 29-136
application data, loading, 17-1
AP Priority, 29-135
APRCEEN, 29-48
APREN, 29-47
APROFEN, 29-47
APRPEEN, 29-48
AP Source Address, 29-136
APTCEN, 29-48
APTSEN, 29-47
Arbitration
  DMA bus, 2-17
  priority for Sys L2 port access request, 2-12
arbitration
  TWI, 23-8
Arbitration priority for Sys L2 port access request, 2-12
ARTS bit, 25-32
asynchronous controller, 1-14
asynchronous FIFO connection, 7-47
asynchronous memory, 5-2
Asynchronous Memory Bank Address Range (table), 5-54
asynchronous memory bank address range (table), 5-5, 5-12, 5-14
Asynchronous Packet Arbitrating (APARB) bit, 29-22
Asynchronous Packet Continuation (APCONT) bit, 29-27
Asynchronous Packet Receive Buffer (APRB), 29-75
Asynchronous Packet Receive Buffer Entry Field Offsets, 29-138
Asynchronous Packet Receive Buffer Entry x (APRBEx) bits, 29-77

Asynchronous Packet Receive Buffer Overflow (APROF) interrupt event, 29-41
Asynchronous Packet Receive Buffer Overflow interrupt enable, 29-45, 29-47
Asynchronous Packet Receive CRC Error (APRCE) interrupt event, 29-42
Asynchronous Packet Receive CRC Error interrupt enable, 29-45, 29-48
Asynchronous Packet Received (APR) bit, 29-23
Asynchronous Packet Received (APR) interrupt event, 29-41
Asynchronous Packet Received interrupt enable, 29-47
Asynchronous Packet Receive Enable (APRXEN) bit, 29-18
Asynchronous Packet Receive Packet Error (APRPE) interrupt event, 29-42
Asynchronous Packet Receive Packet Error interrupt enable, 29-45, 29-48
Asynchronous Packet Receiving (APRX) bit, 29-23
Asynchronous Packet Reception, 29-136
Asynchronous Packet Transmission, 29-134
Asynchronous Packet Transmit Buffer (APTB), 29-75
Asynchronous Packet Transmit Buffer Busy (APBSY) bit, 29-22
Asynchronous Packet Transmit Buffer Field Offsets, 29-135
Asynchronous Packet Transmit Buffer Successfully Cancelled (APTC) interrupt event, 29-42
Asynchronous Packet Transmit Buffer Successfully Cancelled interrupt enable, 29-48
Asynchronous Packet Transmit Buffer Successfully Sent (APTS) interrupt event, 29-42
Asynchronous Packet Transmit Buffer Successfully Sent interrupt enable, 29-47
Asynchronous Packet Transmitting (APTX) bit, 29-22
asynchronous serial communications, 25-6
ASYNC memory banks, 5-3
ATA interface, 21-1
ATAPI
  ATAPI Signals Summary, 21-3
  host DMA state M = machine, 21-12
  host ultra DMA command protocol transfers, 21-15
  PIO data-In state machine, 21-9
  PIO data-put protocol state machine, 21-7
  power-on and hardware reset protocol, 21-19
  summary of IDE/ATA standards, 21-77
ATAPI_ADDR (ATAPI address line status) bits, 21-59
ATAPI address line status (ATAPI_ADDR) bits, 21-59
ATAPI chip select 0 line status (ATAPI_CS0N) bit, 21-59
ATAPI chip select 1 line status (ATAPI_CS1N) bit, 21-59
ATAPI_CONTROL (ATAPI control) register, 21-46, 21-49, A-39
ATAPI_CS0N (ATAPI chip select 0 line status) bit, 21-59
ATAPI_CS1N (ATAPI chip select 1 line status) bit, 21-59
ATAPI_DASP (device DASP to host line status) bit, 21-59

ATAPI_DEV_ADDR (ATAPI device register address) register, 21-46, 21-52, A-39
ATAPI device I/O registers, 21-68
ATAPI device register address (ATAPI_DEV_ADDR) register, 21-46, 21-52, A-39
ATAPI device register receive data (ATAPI_DEV_RXBUF) register, 21-46, 21-54, A-40
ATAPI device register write data (ATAPI_DEV_TXBUF) register, 21-46, 21-53, A-39
ATAPI_DEV_INT (device interrupt status) bit, 21-57
ATAPI_DEV_INT_MASK (device interrupt mask) bit, 21-55
ATAPI_DEV_RXBUF (ATAPI device register receive data) register, 21-46, 21-54, A-40
ATAPI_DEV_TXBUF (ATAPI device register write data) register, 21-46, 21-53, A-39
ATAPI_DIORN (ATAPI read line status) bit, 21-59
ATAPI_DIOWN (ATAPI write line status) bit, 21-59
ATAPI_DMAACKN (ATAPI DMA acknowledge line status) bit, 21-59
ATAPI DMA acknowledge line status (ATAPI_DMAACKN) bit, 21-59
ATAPI_DMAREQ (ATAPI DMA request line status) bit, 21-59
ATAPI DMA request line status (ATAPI_DMAREQ) bit, 21-59
ATAPI_DMA_TFRCNT (ATAPI DMA transfer count) register, 21-47, 21-61, A-40
ATAPI DMA transfer count (ATAPI_DMA_TFRCNT) register, 21-47, 21-61, A-40
ATAPI_HOST_TERM (host termination) bit, 21-60
ATAPI host terminate (ATAPI_TERMINATE) register, 21-47, 21-60, A-40
ATAPI interface, 21-1
ATAPI interrupt mask (ATAPI_INT_MASK) register, 21-46, 21-55, A-40
ATAPI interrupt status (ATAPI_INT_STATUS) register, 21-46, 21-57, A-40
ATAPI_INT_MASK (ATAPI interrupt mask) register, 21-46, 21-55, A-40
ATAPI_INTR (device interrupt to host line status) bit, 21-59
ATAPI_INT_STATUS (ATAPI interrupt status) register, 21-46, 21-57, A-40
ATAPI_IORDY (ATAPI IORDY line status) bit, 21-59
ATAPI IORDY line status (ATAPI_IORDY) bit, 21-59
ATAPI IORDY line status (UDMAOUT_CSTATE) bits, 21-59
ATAPI_LINE_STATUS (ATAPI line status) register, 21-47, 21-59, A-40
ATAPI line status (ATAPI_LINE_STATUS) register, 21-47, 21-59, A-40
ATAPI MDMA timing 0 (ATAPI_MULTI_TIM_0) register, 21-48, 21-65, A-41
ATAPI MDMA timing 1 (ATAPI_MULTI_TIM_1) register, 21-48, 21-65, A-41

ATAPI MDMA timing 2 (ATAPI_MULTI_TIM_2) register, 21-48, 21-66, A-41
ATAPI_MULTI_TIM_0 (ATAPI MDMA timing 0) register, 21-48, 21-65, A-41
ATAPI_MULTI_TIM_1 (ATAPI MDMA timing 1) register, 21-48, 21-65, A-41
ATAPI_MULTI_TIM_2 (ATAPI MDMA timing 2) register, 21-48, 21-66, A-41
ATAPI_PIO_TFRCNT (ATAPI PIO transfer count) register, 21-47, 21-61, A-40
ATAPI_PIO_TIM_0 (ATAPI PIO timing 0) register, 21-47, 21-64, A-41
ATAPI_PIO_TIM_1 (ATAPI PIO timing 1) register, 21-47, 21-64, A-41
ATAPI PIO timing 0 (ATAPI_PIO_TIM_0) register, 21-47, 21-64, A-41
ATAPI PIO timing 1 (ATAPI_PIO_TIM_1) register, 21-47, 21-64, A-41
ATAPI PIO transfer count (ATAPI_PIO_TFRCNT) register, 21-47, 21-61, A-40
ATAPIPI status (ATAPIPI_STATUS) register, 21-62
ATAPI read line status (ATAPI_DIORN) bit, 21-59
ATAPI registers, 21-46, A-39
ATAPI register transfer timing 0 (ATAPI_REG_TIM_0) register, 21-47, 21-63, A-41
ATAPI_REG_TIM_0 (ATAPI register transfer timing 0) register, 21-47, 21-63, A-41
ATAPI_SM_STATE (ATAPI state machine status) register, 21-47, 21-59, A-40
ATAPI standards reference, 21-73
ATAPI state machine status (ATAPI_SM_STATE) register, 21-47, 21-59, A-40
ATAPI_STATUS (ATAPI status) register, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
ATAPI status (ATAPI_STATUS) register, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
ATAPI_STATUS register, 21-46, 21-51, A-39
ATAPI_TERMINATE (ATAPI host terminate) register, 21-47, 21-60, A-40
ATAPI transfer length (ATAPI_XFER_LEN) register, 21-46, 21-58, A-40
ATAPI_UDMAIN_TFRCNT (ATAPI UDMA transfer count) register, 21-47, 21-62, A-40
ATAPI_UDMAOUT_TFRCNT (ATAPI UDMAOUT transfer count) register, 21-47, 21-63, A-41
ATAPI UDMAOUT transfer count (ATAPI_UDMAOUT_TFRCNT) register, 21-47, 21-63, A-41
ATAPI UDMA timing 0 (ATAPI_ULTRA_TIM_0) register, 21-48, 21-66, A-41
ATAPI UDMA timing 1 (ATAPI_ULTRA_TIM_1) register, 21-48, 21-67, A-41
ATAPI UDMA timing 2 (ATAPI_ULTRA_TIM_2) register, 21-48, 21-67, A-41
ATAPI UDMA timing 3 (ATAPI_ULTRA_TIM_3) register, 21-48, 21-68, A-41

ATAPI UDMA transfer count (ATAPI_UDMAIN_TFRCNT) register, 21-47, 21-62, A-40
ATAPI_ULTRA_TIM_0 (ATAPI UDMA timing 0) register, 21-48, 21-66, A-41
ATAPI_ULTRA_TIM_1 (ATAPI UDMA timing 1) register, 21-48, 21-67, A-41
ATAPI_ULTRA_TIM_2 (ATAPI UDMA timing 2) register, 21-48, 21-67, A-41
ATAPI_ULTRA_TIM_3 (ATAPI UDMA timing 3) register, 21-48, 21-68, A-41
ATAPI write line status (ATAPI_DIOWN) bit, 21-59
ATAPI_XFER_LEN (ATAPI transfer length) register, 21-46, 21-58, A-40
atomic operations, 3-77
A_TRANSP (overlay A transparency) bits, 28-41
ATUEN, 29-45
autobaud, and general-purpose timers, 10-33
autobaud detection, 10-33, 25-20
Autobuffer Mode, 29-62, 29-69, 29-73, 29-74
autobuffer mode, 7-18, 7-36, 7-82
AUTOCLEAR_R (RxPktRdy autoclear enable) bit, 26-123
AUTOREQ_RH (autoset ReqPkt) bit, 26-123
autoset ReqPkt (AUTOREQ_R) bit, 26-123
AUTOSET_T (TxPktRdy autoset enable) bit, 26-117
auto-transmit mode, CAN, 31-16
A_VEND (overlay A vertical end) bits, 28-40
A_VSTART (overlay A vertical start) bits, 28-40

B

bable or reset indicator (RESET_OR_BABLE_B) bit, 26-109
bable or reset IRQ enable (RESET_OR_BABLE_BE) bit, 26-110
bank activate command, G-1
BASEID[10:0] field, 31-51, 31-54
baud rate
  SPI, 22-23
  UART, 25-8, 25-19
baud rate[15:0] field, 22-44
BAUD_RATE (baud rate) bits, 22-44
baud rate (BAUD_RATE) bits, 22-44
BCINIT[15:0] field, 7-114
BCOUNT[15:0] field, 7-115
BCZEN, 29-46
B_DEVICE ('A' or 'B' device indicator) bit, 26-134, 26-136
BDI bit, 7-113
BDIE bit, 7-50, 7-113
BDR (burst DMA requests) bit, 8-25, 8-27
BEF bit, 31-85
BFLAG_AUX, 17-126
BFLAG_CALLBACK, 17-126
BFLAG_FASTREAD, 17-125
BFLAG_FILL, 17-126
BFLAG_FINAL, 17-126
BFLAG_FIRST, 17-126
BFLAG_HDRINDIRECT, 17-125
BFLAG_HOOK, 17-125
BFLAG_IGNORE, 17-126
BFLAG_INDIRECT, 17-126
BFLAG_INIT, 17-126
BFLAG_NEXTDXE, 17-125
BFLAG_NOAUTO, 17-125
BFLAG_NONRESTORE, 17-125
BFLAG_PERIPHERAL, 17-125
BFLAG_QUICKBOOT, 17-126
BFLAG_RESET, 17-125

BFLAG_RETURN, 17-125
BFLAG_SAVE, 17-126
BFLAG_SLAVE, 17-125
BFLAG_TYPE, 17-125
BFLAG_WAKEUP, 17-125
B_HEND (overlay B horizontal end) bits, 28-39
B_HSTART (overlay B horizontal start) bits, 28-39
BI (break indicator) bit, 25-36
BI (break interrupt) bit, 25-35
binary decode, B-5
Biphase Mark Coding Error (BMERR) interrupt event, 29-39
Biphase Mark Coding Error interrupt enable, 29-46
bit 15 overflow interrupt enable (COV15IE) bit, 13-28
bit 15 overflow interrupt identifier (COV15II) bit, 13-28
bit 31 overflow interrupt enable (COV31IE) bit, 13-28
bit 31 overflow interrupt identifier (COV31II) bit, 13-28
bit order, selecting, 24-30
BK_DATECODE (Boot Code Date Code), 17-108
BK_DAY, 17-108
BK_ID, 17-107
BK_MONTH, 17-108
BK_ONES, 17-110
BK_PROJECT, 17-107
BK_REVISION (Boot Code Revision Control), 17-107
BK_UPDATE, 17-107
BK_VERSION, 17-107
BK_YEAR, 17-108
BK_ZEROS, 17-109
BL2UEN, 29-45
Blackfin processor family
  I/O memory space, 1-9
  memory architecture, 1-5
BLANKGEN (ITU output with internal blanking) bit, 15-81
block, DMA, 7-16
block code field, 17-27
block count, DMA, 7-46
Block Counter Zero (BCZ) interrupt event, 29-39
Block Counter Zero interrupt enable, 29-46
Block diagram, core, 2-6
block diagrams
  CAN, 31-4
  core timer, 11-2
  DMA controller, 7-6, 7-7
  EBIU, 5-4
  EPPI, 15-5
  general-purpose timers, 10-3
  interrupt processing, 6-24
  PLL, 18-3
  processor, 1-4
  RTC, 14-4
  SPI, 22-3
  SPORT, 24-7
  TWI, 23-3
  UART, 25-3, 25-11
  watchdog timer, 12-3
block done interrupt, DMA, 7-50
Block Flags, 17-29
Block Locked (BLOCK) bit, 29-25
Block Locked to Unlocked (BL2U) interrupt event, 29-35
block transfers, DMA, 7-46
Block Unlocked to Locked (BU2L) interrupt event, 29-35
BMERREN, 29-46
BMODE00_DIS, 17-117
BMODE01_DIS, 17-117

BMODE02_DIS, 17-117
BMODE03_DIS, 17-117
BMODE04_DIS, 17-117
BMODE05_DIS, 17-117
BMODE06_DIS, 17-117
BMODE07_DIS, 17-117
BMODE08_DIS, 17-117
BMODE09_DIS, 17-117
BMODE10_DIS, 17-117
BMODE11_DIS, 17-117
BMODE12_DIS, 17-117
BMODE13_DIS, 17-117
BMODE14_DIS, 17-117
BMODE15_DIS, 17-117
BMODE[2:0] pins, 17-6
BMODE pins, 17-2
BNDMODE (boundary register mode) bits, 13-27
BOIF bit, 31-27, 31-50
BOIM bit, 31-27, 31-49
BOIS bit, 31-27, 31-49
boost PLL amplitude (TM_PLL_VCO) bit, 26-142
boot
  call boot kernel at run time, 17-50
  load function, 17-49
  manager, 17-54
  quick, 17-43
  ROM functions, 17-55
  streams
    multi-DXE, 17-56
Boot Code Date Code (BK_DATECODE), 17-108
Boot Code Revision Control (BK_REVISION), 17-107
boot host wait
  HWAIT, 17-33
booting
  BFROM_MEMBOOT, 17-55
  BFROM_NANDBOOT, 17-55
  BFROM_OTPBOOT, 17-55
  BFROM_SPIBOOT, 17-55
  BFROM_TWIBOOT, 17-55
  boot stream, 17-23
  host boot scenarios, 17-24
  host DMA boot modes, 17-86
  indirect, 17-44
  memory locations, 17-24
  NAND flash boot mode, 17-88
  SPI slave mode, 17-73
  TWI master mode, 17-77
  TWI slave mode, 17-79
booting modes, 17-2
boot kernel, 17-1
Boot Management, 17-54
boot mode
  FIFO boot, 17-67
  flash boot, 17-62
  no-boot, 17-62
  SDRAM boot, 17-66
  SPI device detection, 17-71
boot ROM
  internal, 17-1
boot stream, 17-1, 17-23
boot termination, 17-35
boundary register mode (BNDMODE) bits, 13-27
boundary-scan architecture, B-2
boundary-scan register, B-8
broadcast mode, 22-12, 22-19, 22-20
BRP[9:0] field, 31-10, 31-47
BT_EN (bus timeout enable) bit, 8-25
B_TRANSP (overlay B transparency) bits, 28-41
BU2LEN, 29-45
buffer registers, timers, 10-47

buffers
Cacheability Protection Lookaside Buffers (CPLBs), 3-51, 3-52
Buffer start address register, Initialization, 29-113
BUFRDERR bit, 23-36, 23-38
BUFWRERR bit, 23-36, 23-37
burst DMA requests (BDR) bit, 8-25, 8-27
burst length, G-2
BURST_MODE (DMA burst mode selection) bits, 26-145
burst type, G-2
BUSBUSY bit, 23-36
Bus connection and arbitration, DMA, 2-17
bus contention, avoiding, 19-9
bus error, EBIU, 5-8
BUSERROR (DMA bus error) bit, 26-145
buses
bandwidth, 1-3
and DMA, 7-51
and peripherals, 1-3
prioritization and DMA, 7-58
BUS_MODE (data bus width) bit, 27-56
bus-off interrupt, CAN, 31-27
bus standard, I^2C, 1-15
bus timeout enable (BT_EN) bit, 8-25
B_VEND (overlay B vertical end) bits, 28-40
BV_MULT4 (multiply row by 4) bit, 28-44
B_VSTART (overlay B vertical start) bits, 28-40
BV_TRANS (transparent color - B/V) bits, 28-46
BxMAP (byte x mapping) bits, 9-63
BYPASS bit, 18-26
bypass capacitor placement, 19-16
bypass clock divisor (CLKDIV_BYPASS) bit, 27-56
BYPASS instruction, B-7
bypass register, B-7
byte x mapping (BxMAP) bits, 9-63

C

cache
coherency support, 3-76
mapping into data banks, 3-34
validity of cache lines, 3-16
Cacheability Protection Lookaside Buffers (CPLBs), 3-13, 3-51, 3-52
cache block (definition), 3-79
cache hit, 3-79
address-tag compare, 3-17
data cache access, 3-37
definition, 3-17
processing, 3-17
cache inhibited accesses, 3-77
cache line
components, 3-13
definition, 3-79
states, 3-38
cache miss, 3-79
definition, 3-37
replacement policy, 3-18
Cache way Lock, 3-9
callback routines, 17-45
CAN, 31-1 to 31-92
acceptance mask filtering, 31-17
acceptance mask registers, 31-6
acknowledge error, 31-30
architecture, 31-5
autobaud detection, 10-33
auto-transmit mode, 31-16
bit error, 31-30
bit rate detection, 10-5
bit timing, 31-10
block diagram, 31-4
bus interface, 31-2
clock, 31-10

CAN *(continued)*
code examples, 31-85
configuration mode, 31-10, 31-13
CRC error, 31-30
data field filtering, 31-19
debug and test modes, 31-35
disabling mailboxes, 31-23
enabling mailboxes, 31-87
error frames, 31-28, 31-31
error levels, 31-33
errors, 31-29
event counter, 31-28
extended frame, 31-10
features, 31-2
form error, 31-30
global interrupts, 31-25
hibernate state, 31-40
identifier frame, 31-9
initializing, 31-86
initializing mailboxes, 31-87
initiating transfers, 31-89
interrupts, 31-24, 31-89
lost arbitration, 31-28
and low power designs, 31-40
low power features, 31-38
mailbox area registers, 31-6
mailbox control, 31-7
mailboxes, 31-5
mailbox interrupts, 31-24
mailbox RAM, 31-5
message buffers, 31-5
message received, 31-29
message stored, 31-29
multiplexing of signals, 31-3
nominal bit rate, 31-12
nominal bit time, 31-11
overload frame, 31-28
propagation segment, 31-11
protocol basics, 31-8
receive message lost, 31-28

CAN *(continued)*
receive message rejected, 31-28
receive operation, 31-16
receive operation flow chart, 31-19
registers, table, 31-41
remote frames, 31-22
re-synchronization, 31-12
retransmission, 31-14
sampling, 31-12
single shot transmission, 31-15
sleep mode, 31-39
software reset, 31-13
standard frame, 31-9
stuff error, 31-30
suspend mode, 31-39
test modes, 31-37
time quantum, 31-10
time stamps, 31-21
transceiver interconnection, 31-2
transmission, 31-8
transmission aborted, 31-28
transmission succeeded, 31-28
transmit operation, 31-13
transmit operation flow chart, 31-15
universal counter as event counter, 31-28
valid message, 31-29
wakeup from hibernate, 31-40
warnings, 31-29
watchdog mode, 31-20
CAN_AMxxH (acceptance mask register), 31-51
CAN_AMxxL (acceptance mask register), 31-42, 31-52
CAN_CEC (CAN error counter register), 31-37
Cancel Asynchronous Packet Transmission (CANCELAP) bit, 29-76
Cancel Control Message Transmission (CANCELCM) bit, 29-81

CAN_CLOCK (CAN clock register), 31-10
CAN controller abort acknowledge (CANx_AA1) register 1, 31-76
CAN controller abort acknowledge (CANx_AA2) register 2, 31-76
CAN controller acceptance mask (CANx_AMxxH) registers, 31-50
CAN controller acceptance mask (CANx_AMxxL) registers, 31-50
CAN controller clock (CANx_CLOCK) register, 31-47
CAN controller debug (CANx_DEBUG) register, 31-47
CAN controller error counter (CANx_CEC) register, 31-84
CAN controller error counter warning level (CANx_EWR) register, 31-85
CAN controller error status (CANx_ESR) register, 31-85
CAN controller global interrupt flag (CANx_GIF) register, 31-50
CAN controller global interrupt mask (CANx_GIM) register, 31-49
CAN controller global interrupt status (CANx_GIS) register, 31-49
CAN controller global status (CANx_STATUS) register, 31-46
CAN controller interrupt pending (CANx_INTR) register, 31-48
CAN controller mailbox configuration (CANx_MC1) register 1, 31-69
CAN controller mailbox configuration (CANx_MC2) register 2, 31-69
CAN controller mailbox direction (CANx_MD1) register 1, 31-70
CAN controller mailbox direction (CANx_MD2) register 2, 31-70
CAN controller mailbox interrupt mask (CANx_MBIM1) register 1, 31-79
CAN controller mailbox interrupt mask (CANx_MBIM2) register 2, 31-79
CAN controller mailbox receive interrupt flag (CANx_MBRIF1) register 1, 31-81
CAN controller mailbox receive interrupt flag (CANx_MBRIF2) register 2, 31-81
CAN controller mailbox transmit interrupt flag (CANx_MBTIF1) register 1, 31-80
CAN controller mailbox transmit interrupt flag (CANx_MBTIF2) register 2, 31-80
CAN controller mailbox word 0 (CANx_MBxx_DATA0) register, 31-61
CAN controller mailbox word 1 (CANx_MBxx_DATA1) register, 31-61
CAN controller mailbox word 2 (CANx_MBxx_DATA2) register, 31-61
CAN controller mailbox word 3 (CANx_MBxx_DATA3) register, 31-61
CAN controller mailbox word 4 (CANx_MBxx_LENGTH) register, 31-59
CAN controller mailbox word 5 (CANx_MBxx_TIMESTAMP) register, 31-58
CAN controller mailbox word 6 (CANx_MBxx_ID0) register, 31-56
CAN controller mailbox word 7 (CANx_MBxx_ID1) register, 31-54
CAN controller master control (CANx_CONTROL) register, 31-45

CAN controller overwrite protection/single shot transmission (CANx_OPSS1) register 1, 31-73
CAN controller overwrite protection/single shot transmission (CANx_OPSS2) register 2, 31-73
CAN controller receive message lost (CANx_RML1) register 1, 31-72
CAN controller receive message lost (CANx_RML2) register 2, 31-72
CAN controller receive message pending (CANx_RMP1) register 1, 31-71
CAN controller receive message pending (CANx_RMP2) register 2, 31-71
CAN controller remote frame handling (CANx_RFH1) register 1, 31-78
CAN controller remote frame handling (CANx_RFH2) register 2, 31-78
CAN controller temporary mailbox disable feature (CANx_MBTD) register, 31-78
CAN controller timing (CANx_TIMING) register, 31-48
CAN controller transmission acknowledge (CANx_TA1) register 1, 31-77
CAN controller transmission acknowledge (CANx_TA2) register 2, 31-77
CAN controller transmission request reset (CANx_TRR1) register 1, 31-75
CAN controller transmission request reset (CANx_TRR2) register 2, 31-75
CAN controller transmission request set (CANx_TRS1) register 1, 31-74
CAN controller transmission request set (CANx_TRS2) register 2, 31-74
CAN controller universal counter (CANx_UCCNT) register, 31-84
CAN controller universal counter configuration mode (CANx_UCCNF) register, 31-83
CAN controller universal counter reload/capture (CANx_UCRC) register, 31-84
CAN_DEBUG (CAN debug register), 31-35, 31-36
CAN_EWR (CAN controller error counter warning level) register, 31-45
CAN_MBxx_DATA0 (mailbox word 0 register), 31-43
CAN_MBxx_DATA1 (mailbox word 1 register), 31-43
CAN_MBxx_DATA2 (mailbox word 2 register), 31-43
CAN_MBxx_DATA registers, 31-6
CAN_MBxx_ID0 (mailbox word 6 register), 31-6, 31-42
CAN_MBxx_ID1 (mailbox word 7 register), 31-6
CAN_MBxx_LENGTH (mailbox word 4 register), 31-6
CAN_MBxx_TIMESTAMP (mailbox word 5 register), 31-6
CAN_TIMING (CAN timing register), 31-10
CANWE bit, 18-28
CANx_AA1 (CAN controller abort acknowledge) register 1, 31-76
CANx_AA2 (CAN controller abort acknowledge) register 2, 31-76
CANx_AAx (CAN controller abort acknowledge) registers, 31-43
CANx_AMxxH (CAN controller acceptance mask) registers, 31-42, 31-50
CANx_AMxxL (CAN controller acceptance mask) registers, 31-42, 31-50
CANx_CEC (CAN controller error counter) register, 31-45, 31-84

CANx_CLOCK (CAN controller clock) register, 31-42, 31-47
CANx_CONTROL master control register, 31-42, 31-45
CANx_DEBUG CAN controller debug) register, 31-42, 31-47
CANx_ESR (CAN controller error status) register, 31-45, 31-85
CANx_EWR (CAN controller error counter warning level) register, 31-85
CANx_GIF (CAN controller global interrupt flag) register, 31-42, 31-50
CANx_GIM (CAN controller global interrupt mask) register, 31-42, 31-49
CANx_GIS (CAN controller global interrupt status) register, 31-42, 31-49
CANx_INTR (CAN controller interrupt pending) register, 31-42, 31-48
CANx_MBIM1 (CAN controller mailbox interrupt mask) register 1, 31-79
CANx_MBIM2 (CAN controller mailbox interrupt mask) register 2, 31-79
CANx_MBIMx (CAN controller mailbox interrupt mask) registers, 31-44
CANx_MBRIF1 (CAN controller mailbox receive interrupt flag) register 1, 31-81
CANx_MBRIF2 (CAN controller mailbox receive interrupt flag) register 2, 31-81
CANx_MBRIFx (CAN controller mailbox receive interrupt flag) registers, 31-44
CANx_MBTD (CAN controller temporary mailbox disable feature) register, 31-44, 31-78
CANx_MBTIF1 (CAN controller mailbox transmit interrupt flag) register 1, 31-80
CANx_MBTIF2 (CAN controller mailbox transmit interrupt flag) register 2, 31-80
CANx_MBTIFx (CAN controller mailbox transmit interrupt flag) registers, 31-44
CANx_MBxx_DATA0 (CAN controller mailbox word 0) register, 31-43, 31-61
CANx_MBxx_DATA1 (CAN controller mailbox word 1) register, 31-43, 31-61
CANx_MBxx_DATA2 (CAN controller mailbox word 2) register, 31-43, 31-61
CANx_MBxx_DATA3 (CAN controller mailbox word 3) register, 31-43, 31-61
CANx_MBxx_ID0 (CAN controller mailbox word 6) register, 31-42, 31-56
CANx_MBxx_ID1 (CAN controller mailbox word 7) register, 31-42, 31-54
CANx_MBxx_LENGTH (CAN controller mailbox word 4) register, 31-43, 31-59
CANx_MBxx_TIMESTAMP (CAN controller mailbox word 5) register, 31-42, 31-58
CANx_MC1 (CAN controller mailbox configuration) register 1, 31-69
CANx_MC2 (CAN controller mailbox configuration) register 2, 31-69
CANx_MCx (CAN controller mailbox configuration) registers, 31-43
CANx_MD1 (CAN controller mailbox direction) register 1, 31-70
CANx_MD2 (CAN controller mailbox direction) register 2, 31-70
CANx_MDx (CAN controller mailbox direction) registers, 31-43

CANx_OPSS1 (CAN controller overwrite protection/single shot transmission) register 1, 31-73
CANx_OPSS2 (CAN controller overwrite protection/single shot transmission) register 2, 31-73
CANx_OPSSx (CAN controller overwrite protection/single shot transmission) registers, 31-43
CANx_RFH1 (CAN controller remote frame handling) register 1, 31-78
CANx_RFH2 (CAN controller remote frame handling) register 2, 31-78
CANx_RFHx (CAN controller remote frame handling enable) registers, 31-44
CANx_RML1 (CAN controller receive message lost) register 1, 31-72
CANx_RML2 (CAN controller receive message lost) register 2, 31-72
CANx_RMLx (CAN controller receive message lost) registers, 31-43
CANx_RMP1 (CAN controller receive message pending) register 1, 31-71
CANx_RMP2 (CAN controller receive message pending) register 2, 31-71
CANx_RMPx (CAN controller receive message pending) registers, 31-43
CANxRX bit, 31-48
CANxRX pin, 31-8
CANx_STATUS (CAN controller global status) register, 31-42, 31-46
CANx_TA12 (CAN controller transmission acknowledge) register 2, 31-77
CANx_TA1 (CAN controller transmission acknowledge) register 1, 31-77
CANx_TAx (CAN controller transmit acknowledge) registers, 31-44
CANx_TIMING (CAN controller timing) register, 31-42, 31-48
CANx_TRR1 (CAN controller transmission request reset) register 1, 31-75
CANx_TRR2 (CAN controller transmission request reset) register 2, 31-75
CANx_TRRx (CAN controller transmit request reset) registers, 31-43
CANx_TRS1 (CAN controller transmission request set) register 1, 31-74
CANx_TRS2 (CAN controller transmission request set) register 2, 31-74
CANx_TRSx (CAN controller transmit request set) registers, 31-43
CANxTX bit, 31-48
CANxTX pin, 31-8
CANx_UCCNF (CAN controller universal counter configuration mode) register, 31-44, 31-83
CANx_UCCNT (CAN controller universal counter) register, 31-44, 31-84
CANx_UCRC (CAN controller universal counter reload/capture) register, 31-44, 31-84
capacitors, 19-15
capture mode, *See* WDTH_CAP mode
card detect interrupt enable (SCD_MSK) bit, 27-70
card detect interrupt (SD_CARD_DET) bit, 27-70
CCA bit, 31-46
CCITT G.711 specification, 24-31
CCLK (core clock), 18-4
  disabling, 18-20
  status by operating mode, 18-8

CCR bit, 31-45
CDE bit, 31-35, 31-47
CDGINV (CDG pin polarity invert) bit, 13-27
CDG pin polarity invert (CDGINV) bit, 13-27
CDMAPRIO, 17-106
CFIFO_ERR (chroma FIFO error) bit, 15-80
CFIFO_ERR (Chroma FIFO Overflow Error) bit, 21-49, 21-55, 21-56
Channel-In-Use (CIUx) bit, 29-57
channels
  defined, serial, 24-25
  serial port TDM, 24-25
  serial select offset, 24-25
charge VBUS end interrupt enable (CHRG_VBUS_END_ENA) bit, 26-138
charge VBUS start interrupt enable (CHRG_VBUS_START_ENA) bit, 26-138
CHNL[9:0] field, 24-71, 24-72
CHRG_VBUS_END_ENA (charge VBUS end interrupt enable) bit, 26-138
CHRG_VBUS_START_ENA (charge VBUS start interrupt enable) bit, 26-138
chroma FIFO error (CFIFO_ERR) bit, 15-80
Chroma FIFO Overflow Error (CFIFO_ERR) bit, 21-49, 21-55, 21-56
circuit board testing, B-1, B-6
circular addressing, 7-67
clean (definition), 3-80
clear command response CRC fail (CMD_CRC_FAIL_STAT) bit, 27-65
clear command response received (CMD_RESP_END_STAT) bit, 27-65
clear command sent (CMD_SENT_STAT) bit, 27-65
clear command timeout (CMD_TIMEOUT_STAT) bit, 27-65
clear data block end (DAT_BLK_END_STAT) bit, 27-65
clear data CRC fail (DAT_CRC_FAIL_STAT) bit, 27-65
clear data timeout (DAT_TIMEOUT_STAT) bit, 27-65
CLEAR_DATATOGGLE_R (reset endpoint data toggle) bit, 26-123
CLEAR_DATATOGGLE_T (reset endpoint data toggle) bit, 26-117
clear end of data (DAT_END_STAT) bit, 27-65
clearing interrupt requests, 6-41
clear receive FIFO underrun error (RX_UNDERRUN_STAT) bit, 27-65
clear start bit error (START_BIT_ERR_STAT) bit, 27-65
clear transmit FIFO underrun error (TX_UNDERRUN_STAT) bit, 27-65
CLKBUFOE bit, 18-28
CLKDIV_BYPASS (bypass clock divisor) bit, 27-56
CLKDIV (clock divisor) bits, 26-143, 27-56
CLK_E (SDH_CLK enable) bit, 27-56
CLKHI[7:0] field, 23-26
CLKIN, 1-30, 18-1

CLKIN (input clock), 18-2
CLKIN to VCO, changing the multiplier, 18-14
CLKLOW[7:0] field, 23-26
CLK_SEL bit, 10-13, 10-22, 10-43, 10-51
CLKS_EN (SDH clocks enable) bit, 27-72
clock
  clock signals, 1-30
  EBIU, 5-2
  frequency for SPORT, 24-68
  managing, 19-2
  RTC, 14-3
  source for general-purpose timers, 10-5
  SPI clock signal, 22-5
  types, 19-2
clock divide modulus register, 24-68
clock divisor (CLKDIV) bits, 26-143, 27-56
clock input (CLKIN) pin, 19-2
clock phase, SPI, 22-17, 22-18
clock phase (CPHA) bit, 22-45
clock polarity, SPI, 22-17
clock polarity (CPOL) bit, 22-45
clock rate
  core timer, 11-1
  SPORT, 24-2
clock ratio, changing, 18-6
clocks
  internal, 2-5
CM Allocate Channel List field, 29-151
CM Allocate Number Free field, 29-151
CM Allocate Number Requested field, 29-149
CM Allocate Status, 29-151
CM Allocate Status Encodings, 29-151
CMD_ACT (command active) bit, 27-64
CMD_ACT_MASK (command active) bit, 27-67
CM Data field, 29-143
CMD_CRC_FAIL (command response CRC fail) bit, 27-64
CMD_CRC_FAIL_MASK (command response CRC fail) bit, 27-67
CMD_CRC_FAIL_STAT (clear command response CRC fail) bit, 27-65
CM De-Allocate Connection Label field, 29-152
CM De-Allocate Status, 29-154
CM De-Allocate Status Encodings, 29-154
CMD_E (command enable) bit, 27-58
CM Destination Address, 29-140
CMD_IDX (command index) bits, 27-58
CMD_INT_E (command interrupt enable) bit, 27-58
CMD_L_RSP (long response enable) bit, 27-58
CMD_PEND_E (pend enable) bit, 27-58
CMD_RESP_END (command response received) bit, 27-64
CMD_RESP_END_MASK (command response received) bit, 27-67
CMD_RESP_END_STAT (clear command response received) bit, 27-65
CMD_RSP (wait for response) bit, 27-58
CMD_SENT (command sent) bit, 27-64
CMD_SENT_MASK (command sent) bit, 27-67
CMD_SENT_STAT (clear command sent) bit, 27-65
CMD_TIMEOUT (command time out) bit, 27-64
CMD_TIMEOUT_MASK (command timeout) bit, 27-67
CMD_TIMEOUT_STAT (clear command timeout) bit, 27-65
CM GetSource Channel field, 29-155
CM Message Type Encodings, 29-140

CM Message Type field, 29-140
CM Priority, 29-139
CM Read Address field, 29-144
CMREN, 29-45
CM Source Address, 29-140
CMTCEN, 29-46
CM Transmission Status, 29-141
CM Transmission Status field, 29-140, 29-141
CMTSEN, 29-45
CM Write Address field, 29-146
CM Write Data field, 29-147
CM Write Length field, 29-146
CNFG_PEND (config pending) bit, 8-27
CNOS (tuning of DPHY clocks) bits, 26-141
CNT_COMMAND (command) register, 13-29
CNT_CONFIG (configuration) register, 13-27, A-91
CNT_COUNTER (counter) register, 13-25, 13-31
CNT_DEBOUNCE (debounce) register, 13-25, 13-31
CNTE (counter enable) bit, 13-27
CNT_IMASK (interrupt mask) register, 13-25, 13-28, A-91, A-92
CNT_MAX (maximal count) register, 13-25, 13-32
CNT_MIN (minimal count) register, 13-25, 13-32
CNTMODE (counter operating mode) bits, 13-27
CNT_STATUS (status) register, 13-28
codecs, connecting to, 24-2
code examples
  CSYNC, 3-79
  interrupt enabling and disabling, 3-79
  load base of MMRs, 3-79
  restoration of the control register, 3-79
COLDRV_SCALE (column driver scale value) bits, 30-15
column address
  strobe latency, G-4
column driver scale value (COLDRV_SCALE) bits, 30-15
column enable width (KPAD_COLEN) bits, 30-10
columns value pressed (KPAD_COL) bits, 30-15
command active (CMD_ACT) bit, 27-64
command active (CMD_ACT_MASK) bit, 27-67
command (CNT_COMMAND) register, 13-29
command enable (CMD_E) bit, 27-58
command index (CMD_IDX) bits, 27-58
command index of last received response (RESP_CMD) bits, 27-59
command interrupt enable (CMD_INT_E) bit, 27-58
command response CRC fail (CMD_CRC_FAIL) bit, 27-64
command response CRC fail (CMD_CRC_FAIL_MASK) bit, 27-67
command response received (CMD_RESP_END) bit, 27-64
command response received (CMD_RESP_END_MASK) bit, 27-67
commands
  bank activate, G-1
  DMA control, 7-39, 7-40
  precharge, G-18
  transfer initiate, 22-27
command sent (CMD_SENT) bit, 27-64
command sent (CMD_SENT_MASK) bit, 27-67

command time out (CMD_TIMEOUT) bit, 27-64
command timeout (CMD_TIMEOUT_MASK) bit, 27-67
companding, 24-17, 24-26
  defined, 24-31
  lengths supported, 24-31
  multichannel operations, 24-26
COMPLETE (DMA complete) bit, 8-27
conditional
  branches, 3-74
config pending (CNFG_PEND) bit, 8-27
configuration
  CAN, 31-13
  L1 Instruction Memory, 3-13
  L1 SRAM, 3-2
  precautions before changing, 3-12
  SPORT, 24-12
configuration (CNT_CONFIG) register, 13-27, A-91
congestion, on DMA channels, 7-55
CONN_B (connection indicator) bit, 26-109
CONN_BE (connection IRQ enable) bit, 26-110
Connection and arbitration, DMA bus, 2-17
connection indicator (CONN_B) bit, 26-109
connection IRQ enable (CONN_BE) bit, 26-110
Connection Label, 29-152
Connection Label (CLx) field, 29-56
Content-Addressable Memory (CAM), 3-51
continuous transition, DMA, 7-35
control bit summary, general-purpose timers, 10-51
Controller, DMA, 2-4, 2-18
Control Message Arbitrating (CMARB) bit, 29-23
Control Message Interrupt, 29-30
Control Message Receive Buffer, 29-16
Control Message Receive Buffer (CMRB), 29-80
Control Message Receive Buffer Entry Field Offsets, 29-159
Control Message Receive Buffer Entry Offsets, 29-158
Control Message Receive Buffer Entry x (CMRBEx) bits, 29-81
Control Message Receive Buffer Overflow (CMROF) interrupt event, 29-38
Control Message Receive Buffer Overflow interrupt enable, 29-45
Control Message Received (CMR) bit, 29-24
Control Message Received (CMR) interrupt event, 29-37
Control Message Received interrupt enable, 29-45
Control Message Reception, 29-157
Control Message Transmission, 29-138
Control Message Transmit Buffer Busy (CMBSY) bit, 29-23
Control Message Transmit Buffer (CMTB), 29-80
Control Message Transmit Buffer Successfully Cancelled (CMTC) interrupt event, 29-38
Control Message Transmit Buffer Successfully Cancelled interrupt enable, 29-46
Control Message Transmit Buffer Successfully Sent (CMTS) interrupt event, 29-38
Control Message Transmit Buffer Successfully Sent interrupt enable, 29-45

Control Message Transmitting (CMTX) bit, 29-24
Control/optimization, DMA traffic, 2-13
control register
  data memory, 3-27
  restoration, 3-79
core
  core clock (CCLK), 18-4, 19-2
  core clock/system clock ratio control, 18-4
  powering down, 18-20
  waking from idle state, 6-13
core and system reset, code example, 8-30, 17-145, 17-146
Core block diagram, 2-6
core clock, *See* CCLK
core double-fault reset, 17-6
core event controller (CEC), 6-2, 6-6
core-only software reset, 17-6
core timer, 11-1 to 11-8
  block diagram, 11-2
  clock rate, 11-1
  features, 11-1
  initialization, 11-3
  internal interfaces, 11-2
  interrupts, 11-3
  low power mode, 11-3
  operation, 11-3
  registers, 11-4
  scaling, 11-7
core timer control register (TCNTL), 11-3, 11-5
core timer count register (TCOUNT), 11-5
core timer period register (TPERIOD), 11-6
core timer scale register (TSCALE), 11-7
counter, RTC, 14-2
counter (CNT_COUNTER) register, 13-25, 13-31
counter enable (CNTE) bit, 13-27
counter operating mode (CNTMODE) bits, 13-27
Count Position (COUNTPOSx) field, 29-67
COUNT_TIMEOUT (host timeout count) bits, 8-29
count to zero interrupt enable (CZEROIE) bit, 13-28
count to zero interrupt identifier (CZEROII) bit, 13-28
count value[15:0] field, 11-6
count value[31:16] field, 11-6
COV15IE (bit 15 overflow interrupt enable) bit, 13-28
COV15II (bit 15 overflow interrupt identifier) bit, 13-28
COV31IE (bit 31 overflow interrupt enable) bit, 13-28
COV31II (bit 31 overflow interrupt identifier) bit, 13-28
CPHA bit, 22-45
CPHA (clock phase) bit, 22-45
CPOL bit, 22-45
CPOL (clock polarity) bit, 22-45
CRC32 checksum generation, 17-48
CRCE bit, 31-85
CRC error, 29-144
CROSSCORE software, 1-36
crosstalk, 19-15
crystal oscillator pins, 29-2
CSA bit, 31-39, 31-46
CSEL[1:0] field, 18-5, 18-26
CSR bit, 31-39, 31-45
CSR_HBR (USB hibernate signal) bit, 26-141
CSR_RSTD (USB pu/pd restore control) bit, 26-141
CSYNC, 3-74
  code example, 3-79

CTS (clear to send) bit, 25-37
CTYPE bit, 7-77
CUD and CDZ input disable (INPDIS) bit, 13-27
CUDINV (CUD pin polarity invert) bit, 13-27
CUD pin polarity invert (CUDINV) bit, 13-27
current address field, 7-91
current address registers
  (DMAx_CURR_ADDR), 7-90
  (MDMA_yy_CURR_ADDR), 7-90
current descriptor pointer field, 7-109
current descriptor pointer registers
  (DMAx_CURR_DESC_PTR), 7-108
  (MDMA_yy_CURR_DESC_PTR), 7-108
current inner loop count registers
  (DMAx_CURR_X_COUNT), 7-94, 7-95
  (MDMA_yy_CURR_X_COUNT), 7-94, 7-95
current outer loop count registers
  (DMAx_CURR_Y_COUNT), 7-101
  (MDMA_yy_CURR_Y_COUNT), 7-101
CURR_X_COUNT[15:0] field, 7-95
CURR_Y_COUNT[15:0] field, 7-102
CZEROIE (count to zero interrupt enable) bit, 13-28
CZEROII (count to zero interrupt identifier) bit, 13-28
CZMEIE (CZM error interrupt enable) bit, 13-28
CZMEII (CZM error interrupt identifier) bit, 13-28
CZM error interrupt enable (CZMEIE) bit, 13-28
CZM error interrupt identifier (CZMEII) bit, 13-28
CZMIE (CZM pin interrupt enable) bit, 13-28
CZMII (CZM pin interrupt identifier) bit, 13-28
CZMINV (CZM pin polarity invert) bit, 13-27
CZM pin interrupt enable (CZMIE) bit, 13-28
CZM pin interrupt identifier (CZMII) bit, 13-28
CZM pin polarity invert (CZMINV) bit, 13-27
CZM zeroes counter enable (ZMZC) bit, 13-27
CZM zeroes counter interrupt enable (CZMZIE) bit, 13-28
CZM zeroes counter interrupt identifier (CZMZII) bit, 13-28
CZMZIE (CZM zeroes counter interrupt enable) bit, 13-28
CZMZII (CZM zeroes counter interrupt identifier) bit, 13-28

D

DAB, 7-51, 7-120
  clocking, 18-1
DAB, DMA Access Bus, 2-5, 2-17
DAB_TRAFFIC_COUNT[2:0] field, 7-120
data block end (DAT_BLK_END) bit, 27-64
data block end (DAT_BLK_END_MASK) bit, 27-67
data bus width (WIDE_BUS) bit, 27-56
data cache control instructions, 3-41
data corruption, avoiding with SPI, 22-19
DATA_COUNT (data count) bits, 27-62
data count (DATA_COUNT) bits, 27-62
data CRC fail (DAT_CRC_FAIL) bit, 27-64

data CRC fail (DAT_CRC_FAIL_MASK) bit, 27-67
data-driven interrupts, 7-87
DATAEND (data end indicator) bit, 26-113
data end (DAT_END) bit, 27-64
data end indicator (DATAEND) bit, 26-113
DATAERROR_R (load error indicator) bit, 26-123
data field byte 0[7:0] field, 31-61
data field byte 1[7:0] field, 31-61
data field byte 2[7:0] field, 31-63
data field byte 3[7:0] field, 31-63
data field byte 4[7:0] field, 31-65
data field byte 5[7:0] field, 31-65
data field byte 6[7:0] field, 31-66
data field byte 7[7:0] field, 31-66
data field filtering, CAN, 31-19
data formats, SPORT, 24-30
data interrupt timing select (DI_SEL) bit, 8-6
DATA_LENGTH (number of bytes to transfer) bits, 27-61
data memory, L1, 3-27
Data Memory Control register (DMEM_CONTROL), 3-27, 3-52
data move, serial port operations, 24-40
data operations, CPLB, 3-52
data packet in FIFO indicator (FIFO_NOT_EMPTY_T) bit, 26-117
data packet in FIFO indicator (RXPKTRDY_R) bit, 26-123
data packet in FIFO indicator (TXPKTRDY) bit, 26-113
data packet in FIFO indicator (TXPKTRDY_T) bit, 26-117
data packet receive indicator (RXPKTRDY) bit, 26-113
data receive active (RX_ACT) bit, 27-64
data receive active (RX_ACT_MASK) bit, 27-67
data sampling, serial, 24-35
DATA_SIZE bit, 8-25
Data SRAM
  L1, 3-30
data store format, 3-80
data structures, 17-120
  boot_struct, 17-122
  buffer_struct, 17-121
  header_struct, 17-120
Data Test Command register (DTEST_COMMAND), 3-44
Data Test Data registers (DTEST_DATAx), 3-45
data timeout bits, 27-60
data time out (DAT_TIMEOUT bit, 27-64
data timeout (DAT_TIMEOUT_MASK) bit, 27-67
data transfer block length (DTX_BLK_LGTH) bits, 27-62
data transfer direction (DTX_DIR) bit, 27-62
data transfer DMA enable (DTX_DMA_E) bit, 27-62
data transfer enable (DTX_E) bit, 27-62
Data transfer latency
  DMA, 2-13
data transfer mode (DTX_MODE) bit, 27-62
data transfers
  SPI, 22-20
data transmit active (TX_ACT) bit, 27-64
data transmit active (TX_ACT_MASK) bit, 27-67
data word, serial data formats, 24-61
DAT_BLK_END (data block end) bit, 27-64

DAT_BLK_END_MASK (data block end) bit, 27-67
DAT_BLK_END_STAT (clear data block end) bit, 27-65
DAT_CRC_FAIL (data CRC fail) bit, 27-64
DAT_CRC_FAIL_MASK (data CRC fail) bit, 27-67
DAT_CRC_FAIL_STAT (clear data CRC fail) bit, 27-65
DAT_END (data end) bit, 27-64
DAT_END_MASK (end of data) bit, 27-67
DAT_END_STAT (clear end of data) bit, 27-65
DAT_TIMEOUT (data time out) bit, 27-64
DAT_TIMEOUT_MASK (data timeout) bit, 27-67
DAT_TIMEOUT_STAT (clear data timeout) bit, 27-65
day[14:0] field, 14-23
day alarm interrupt enable bit, 14-21
day counter[14:0] field, 14-21
DBON_SCALE (debounce scale value) bits, 30-15
DCB, 7-51, 7-121
  DCB1, 2-10
  DCB2, 2-10, 2-11
DCB, DMA Core Bus, 2-5, 2-17
DCB bus arbitration, 2-18
DCBS (L1 Data Cache Bank Select) bit, 3-35
DCB_TRAFFIC_COUNT field, 7-121
DCB_TRAFFIC_PERIOD field, 7-121
DCIE (down count interrupt enable) bit, 13-28
DCII (down count interrupt identifier) bit, 13-28
DCNT[7:0] field, 23-32, 23-33
DCPLB Address registers (DCPLB_ADDRx), 3-63
DCPLB_ADDRx (DCPLB Address registers), 3-63
DCPLB Data registers (DCPLB_DATAx), 3-61
DCPLB_DATAx (DCPLB Data registers), 3-61
DCPLB_FAULT_ADDR (DCPLB Fault Address register), 3-67
DCPLB Fault Address register (DCPLB_FAULT_ADDR), 3-67
DCPLB_STATUS (DCPLB Status register), 3-66
DCPLB Status register (DCPLB_STATUS), 3-66
DDR, 2-2
DDR SDRAM controller, 1-13
DEB, 7-51, 7-121
DEB, DMA External Bus, 2-5, 2-17
DEBE (debounce enable) bit, 13-27
debounce (CNT_DEBOUNCE) register, 13-25, 13-31
debounce enable (DEBE) bit, 13-27
debounce scale value (DBON_SCALE) bits, 30-15
DEB_TRAFFIC_COUNT field, 7-121
DEB_TRAFFIC_PERIOD field, 7-121
debugging
  test point access, 19-18
DEC bit, 31-37, 31-47
deep sleep mode, 1-32, 18-10
default mapping, peripheral to DMA, 7-10
Delay Register Updated (DRU) interrupt event, 29-32
Delay Register Updated interrupt enable, 29-44
DERREN, 29-46
descriptor array mode, DMA, 7-22, 7-82
descriptor-based DMA, 7-21

descriptor chains, DMA, 7-35
descriptor list mode, DMA, 7-22, 7-82, 7-83
descriptor queue, 7-67
  management, 7-67
  synchronization, 7-67, 7-68
descriptor structures
  DMA, 7-65
  MDMA, 7-72
destination channels, memory DMA, 7-13
detected FIFO not empty (FIFO_FULL_R) bit, 26-123
DEV_ADDR (device address) bit, 21-52
development tools, 1-36
device address (DEV_ADDR) bit, 21-52
device DASP to host line status (ATAPI_DASP) bit, 21-59
device interrupt mask (ATAPI_DEV_INT_MASK) bit, 21-55
device interrupt status (ATAPI_DEV_INT) bit, 21-57
device interrupt to host line status (ATAPI_INTR) bit, 21-59
device receive buffer (REG_RXBUFFER) bits, 21-54
device terminate multi-DMA transfer interrupt mask (MULTI_TERM_MASK) bit, 21-55
device terminate multi-DMA transfer interrupt status (MULTI_TERM_INT) bit, 21-57
device terminate ultra-DMA-in transfer interrupt mask (UDMAIN_TERM_MASK) bit, 21-55
device terminate ultra-DMA-in transfer interrupt status (UDMAIN_TERM_INT) bit, 21-57
device terminate ultra-DMA-out transfer interrupt mask (UDMAOUT_TERM_MASK) bit, 21-55
device terminate ultra-DMA-out transfer interrupt status (UDMAOUT_TERM_INT) bit, 21-57
device transmit buffer (REG_TXBUFFER) bits, 21-53
DEV_RST (device reset) bit, 21-49
DF bit, 18-3, 18-4, 18-26
DFC[15:0] field, 31-56
DF (divide CLKIN by 2) bit, 26-142
DFETCH bit, 7-22, 7-29, 7-85
dFlags Word, Bits 15–0, 17-126
dFlags Word, Bits 31–16, 17-125
DFM[15:0] field, 31-52
DFRESET, 16-58, 16-59, 17-106
DI_EN bit, 7-21, 7-80, 7-83
DIL bit, 31-36, 31-47
DIOR/DIOW asserted pulsewidth (TD) bits, 21-65
DIOR/DIOW pulsewidth (T2_REG_PIO) bits, 21-64
DIOW data hold (T4_REG) bits, 21-64
DIR (direction) bit, 15-81
direct code execution, 17-37
  initial header, 17-36, 17-38
DIRECTION (DMA Tx or Rx selection) bit, 26-145
direct mapped (definition), 3-79
direct memory access, *See* DMA
dirty (definition), 3-80
Disable Interrupts (CLI) instruction, 3-79
disable nyet handshake (DISNYET) bit, 26-123
disabling
  general-purpose timers, 10-39
  PLL, 18-13

discharge VBUS end interrupt enable (DISCHRG_VBUS_END_ENA) bit, 26-138
discharge VBUS start interrupt enable (DISCHRG_VBUS_START_ENA) bit, 26-138
DISCHRG_VBUS_END_ENA (discharge VBUS end interrupt enable) bit, 26-138
DISCHRG_VBUS_START_ENA (discharge VBUS start interrupt enable) bit, 26-138
DISCON_B (disconnect/session end indicator) bit, 26-109
DISCON_BE (disconnect/session end IRQ enable) bit, 26-110
disconnect/session end indicator (DISCON_B) bit, 26-109
disconnect/session end IRQ enable (DISCON_BE) bit, 26-110
DI_SEL bit, 7-80, 7-83
DI_SEL (data interrupt, 8-6
DISNYET (disable nyet handshake) bit, 26-123
DITFS bit, 24-40, 24-51, 24-55, 24-66
divide CLKIN by 2 (DF) bit, 26-142
divisor latch high byte[15:8] field, 25-48
divisor latch low byte[7:0] field, 25-48
divisor reset, UART, 25-49
DLC[3:0] field, 31-59
DLEN (data length) bits, 15-82
DMA, 7-1 to 7-133
  1D interrupt-driven, 7-64
  1D unsynchronized FIFO, 7-65
  2D, polled, 7-64
  2D array, example, 7-123
  2D interrupt-driven, 7-64
  autobuffer mode, 7-18, 7-36, 7-82
  bandwidth, 7-55
  block count, 7-46
DMA *(continued)*
  block diagram, 7-6, 7-7
  block done interrupt, 7-50
  block transfers, 7-16, 7-46
  buffer size, multichannel SPORT, 24-26
  channel registers, 7-76
  channels, 7-51
  channels and control schemes, 7-60
  channel-specific register names, 7-75
  congestion, 7-55
  connecting asynchronous FIFO, 7-47
  continuous transfers using autobuffering, 7-64
  continuous transition, 7-35
  control command restrictions, 7-43
  control commands, 7-39, 7-40
  data transfers, 7-2
  default peripheral mapping, 7-10
  descriptor array, 7-30
  descriptor array mode, 7-22, 7-82
  descriptor-based, 7-21
  descriptor-based, initializing, 7-126
  descriptor-based vs. register-based transfers, 7-3
  descriptor chains, 7-35
  descriptor element offsets, 7-23
  descriptor list mode, 7-22, 7-82, 7-83
  descriptor lists, 7-30
  descriptor queue, 7-67
  descriptors, recommended size, 7-24
  descriptor structures, 7-65
  direction, 7-84
  DMA error interrupt, 7-88
  double buffer scheme, 7-64
  and EBIU, 7-8
  errors, 7-37, 7-38
  example connection, receive, 7-49
  example connection, transmit, 7-48
  external interfaces, 7-8
  finish control command, 7-41

DMA *(continued)*
first data memory access, 7-29
flow chart, 7-26, 7-27
FLOW mode, 7-24
FLOW value, 7-28
for SPI transmit, 22-14
functions, summary, 7-4
handshake operation, 7-45
header file to define descriptor structures example, 7-127
HMDMA1 block enable example, 7-132
initializing, 7-25
internal interfaces, 7-8
large model mode, 7-83
latency, 7-32
mapping to peripherals, 6-14, 6-15
memory conflict, 7-59
memory DMA, 7-13
memory DMA transfers, 7-9
memory read, 7-33
operation flow, 7-25
orphan access, 7-36
overflow interrupt, 7-50
overview, 1-10
performance considerations, 7-52
peripheral, 7-10
peripheral channels, 7-2
peripheral channels priority, 7-11
peripheral interrupts, 6-13
pipelining requests, 7-47
polling DMA status example, 7-125
polling registers, 7-61
prioritization and traffic control, 7-54 to 7-60
programming examples, 7-122 to 7-133
receive, 7-35
receive restart or finish, 7-44
refresh, 7-30
register-based, 7-17

DMA *(continued)*
register-based 2D memory DMA example, 7-123
register naming conventions, 7-76
remapping peripheral assignment, 7-11
request data control command, 7-42
request data urgent control command, 7-42
restart control command, 7-40
round robin operation, 7-57
serial port block transfers, 24-40
single-buffer transfers, 7-63
small model mode, 7-82
software management, 7-60
software-triggered descriptor fetch example, 7-129
and SPI, 22-14
SPI data transmission, 22-15, 22-16
SPI master, 22-33
SPI slave, 22-35
and SPORT, 24-3
startup, 7-25
stop mode, 7-18, 7-82
stopping transfers, 7-36
support for peripherals, 1-3
switching peripherals from, 7-88
synchronization, 7-60 to 7-70
synchronized transition, 7-35
termination without abort, 7-36
throughput, 7-51
traffic control, 7-58
traffic exceeding available bandwidth, 7-55
transfers, urgent, 7-54
transmit, 7-33
transmit restart or finish, 7-44
triggering transfers, 7-71
two descriptors in small list flow mode, example, 7-126
two-dimensional, 7-19

DMA *(continued)*
two-dimensional memory DMA setup example, 7-124
and UART, 25-24, 25-41
using descriptor structures example, 7-128
variable descriptor size, 7-23
word size, changing, 7-35, 7-36
work units, 7-21, 7-30, 7-32
DMA, Related Buses, 2-17
DMA0 Urgent Request (DMA0URQ) bit, 21-49, 21-55, 21-56
DMA0URQ (DMA0 Urgent Request) bit, 21-49, 21-55, 21-56
DMA0URQ (DMA0 urgent request) bit, 15-80
DMA1 Urgent Request (DMA1URQ) bit, 21-49, 21-55, 21-56
DMA1URQ (DMA1 Urgent Request) bit, 21-49, 21-55, 21-56
DMA1URQ (DMA1 urgent request) bit, 15-80
DMA2D bit, 7-80, 7-83
DMA2D (DMA mode) bit, 8-6
DMA Access Bus (DAB), 2-5, 2-17
DMA access to L1 or L2 memory, stalls, 2-13
DMAACTIVEx bits, 29-28
DMA address high (DMA_ADDR_HIGH) bits, 26-147
DMA address low (DMA_ADDR_LOW) bits, 26-146
DMA_ADDR_HIGH (DMA address high) bits, 26-147
DMA_ADDR_LOW (DMA address low) bits, 26-146
DMA burst mode selection (BURST_MODE) bits, 26-145
DMA Bus
connection and arbitration, 2-17
DMA bus error (BUSERROR) bit, 26-145
DMAC1_PERIMUX (DMA controller 1 peripheral multiplexer) register, 7-121
DMACFG field, 7-29, 7-72
DMACFG (one/two DMA channel modes) bit, 15-82
DMA channel registers, 7-73
DMA Channel x Done interrupt enable, 29-47
DMA Channel x Half Done interrupt enable, 29-47
DMACODE, 17-126
DMA Code field
DMACODE, 17-27
DMA complete (COMPLETE) bit, 8-27
DMA configuration registers
(DMAx_CONFIG), 7-79
(MDMA_yy_CONFIG), 7-79
DMA controller, 2-4, 2-18, 7-2
DMA Core Bus (DCB), 2-5, 2-17
DMA_COUNT_LOW (DMA count low) bits, 26-147, 26-148
DMA count low (DMA_COUNT_LOW) bits, 26-147, 26-148
DMA_CSTATE (DMA mode state machine current state) bits, 21-59
DMACx_TCCNT (DMA traffic control counter) registers, 7-119
DMACx_TCPER (DMA traffic control counter period) registers, 7-119
DMA data transfer latency, 2-13
DMA_DIR (DMA direction) bit, 8-27
DMA direction (DMA_DIR) bit, 8-27
DMA_DONE bit, 7-85
DMA_DONE interrupt, 7-85
DMA enable (DMA_ENA) bit, 26-145
DMA_ENA (DMA enable) bit, 26-145
DMAEN bit, 7-25, 7-71, 7-80, 7-84
DMA_ERR bit, 7-85

DMA Error Channel Number (DERRNUM) field, 29-26
DMA Error (DERR) interrupt event, 29-39
DMA_ERROR interrupt, 7-37
DMA Error interrupt enable, 29-46
DMA error interrupts, 7-87
DMA External Bus (DEB), 2-5, 2-17
DMA mode 0/1 selection (MODE) bit, 26-145
DMA mode select (DMAREQMODE_R) bit, 26-123
DMA mode select (DMAREQMODE_RH) bit, 26-123
DMA mode select (DMAREQMODE_T) bit, 26-117
DMA mode state machine current state (DMA_CSTATE) bits, 21-59
DMA performance optimization, 7-50
DMAPMENx, 29-29
DMA queue completion interrupt, 7-70
DMAR0 pin, 7-8
DMAR1 pin, 7-8
DMA ready (READY) bit, 8-27
DMA registers, 7-73
DMAREQ_ENA_R (DMA request enable) bit, 26-123
DMAREQ_ENA_T (DMA request enable) bit, 26-117
DMAREQMODE_R (DMA mode select) bit, 26-123
DMAREQMODE_RH (DMA mode select) bit, 26-123
DMAREQMODE_T (DMA mode select) bit, 26-117
DMA request enable (DMAREQ_ENA_R) bit, 26-123
DMA request enable (DMAREQ_ENA_T) bit, 26-117
DMA_RUN bit, 7-29, 7-68, 7-72, 7-85
DMARx pin, 7-47
DMA start address field, 7-89
DMA_TC_CNT (DMA traffic control counter register), 7-120
DMA_TC_PER (DMA traffic control counter period register), 7-57, 7-120
DMA traffic control counter period register (DMA_TC_PER), 7-120
DMA traffic control counter register (DMA_TC_CNT), 7-120
DMA traffic control/optimization, 2-13
DMA_TRAFFIC_PERIOD field, 7-120
DMA Tx or Rx selection (DIRECTION) bit, 26-145
DMAx Bit-Swap Enable (BITSWAPENx) bit, 29-61
DMAx_CONFIG (DMA configuration registers), 7-15, 7-25, 7-33, 7-79
DMAx_CURR_ADDR (current address registers), 7-90
DMAx_CURR_DESC_PTR (current descriptor pointer registers), 7-108
DMAx_CURR_X_COUNT (current inner loop count registers), 7-94, 7-95
DMAx_CURR_Y_COUNT (current outer loop count registers), 7-101
DMAx Direction (DDx) bit, 29-60
DMAx Done (DONEx) interrupt event, 29-41
DMAx Half-Done (HDONEx) interrupt event, 29-41
DMAx_INT (USB DMA endpoint x interrupt) bits, 26-144
DMAx_IRQ_STATUS (interrupt status registers), 7-84, 7-85
DMAx Logical Channel (LCHANx) field, 29-60
DMAx_NEXT_DESC_PTR (next descriptor pointer registers), 7-25, 7-106

DMAx Operation Flow (MFLOWx) field, 29-62
DMAx_PERIPHERAL_MAP (peripheral map registers), 7-77
DMAx_START_ADDR (start address registers), 7-25, 7-88
DMAx_X_COUNT (inner loop count registers), 7-92
DMAx_X_MODIFY (inner loop address increment registers), 7-25, 7-97
DMAx_Y_COUNT (outer loop count registers), 7-99
DMAx_Y_MODIFY (outer loop address increment registers), 7-25, 7-103
DMEM_CONTROL (Data Memory Control register), 3-27, 3-52
DNAK bit, 23-36, 23-38
DNM bit, 31-45
DONEENx, 29-47
DOUBLE_FAULT, 17-104
DOUBLE_FAULT bit, 16-40
DOUBLE_RESET, 16-40
DPMC, 18-2, 18-7 to 18-11
D Port, 2-4
  interface, 2-9
DR bit, 25-17
DR (data ready) bit, 25-35
DR flag, 25-23
DRI bit, 31-37, 31-47
DRIVE_VBUS_OFF_ENA (drive VBUS off interrupt enable) bit, 26-138
drive VBUS off interrupt enable (DRIVE_VBUS_OFF_ENA) bit, 26-138
DRIVE_VBUS_ON_ENA (drive VBUS on interrupt enable) bit, 26-138
drive VBUS on interrupt enable (DRIVE_VBUS_ON_ENA) bit, 26-138
DRQ[1:0] field, 7-55, 7-112, 7-113
DRUEN, 29-44
DRxPRI signal, 24-6
DRxPRI SPORT input, 24-8
DRxSEC signal, 24-6
DRxSEC SPORT input, 24-8
DTEST_COMMAND (Data Test Command register), 3-44
DTEST_DATAx (Data Test Data registers), 3-45
DTO bit, 31-36, 31-47
DTX_BLK_LGTH (data transfer block length) bits, 27-62
DTX_DIR (data transfer direction) bit, 27-62
DTX_DMA_E (data transfer DMA enable) bit, 27-62
DTX_E (data transfer enable) bit, 27-62
DTX_MODE (data transfer mode) bit, 27-62
DTxPRI signal, 24-6
DTxPRI SPORT output, 24-8
DTxSEC signal, 24-6
DTxSEC SPORT output, 24-8
dynamic power management, 1-31, 18-1
dynamic power management controller, 18-2

E

EAB
  clocking, 18-1
EAB, External Access Bus, 2-4, 2-5, 2-24
early frame sync, 24-38
EBIU, 1-13, 5-1
  as slave, 5-7
  block diagram, 5-4
  bus errors, 5-8
  clock, 5-2
  and DMA, 7-8
  error detection, 5-8
  overview, 5-1

EBIU *(continued)*
request priority, 5-2
EBIU, External Bus Interface Unit, 2-2
EBIU_AMGCTL (Asynchronous Memory Global Control register), 5-58
EBIU Pin List (with Multiplexing), 5-5
EBO bit, 31-46
ECC0 (parity calculation result0) bits, 20-23
ECC1 (parity calculation result1) bits, 20-23
ECC2 (parity calculation result2) bits, 20-23
ECC3 (parity calculation result3) bits, 20-23
ECCCNT (transfer count) bits, 20-24
ECC_RST (ECC (and NFC counters) reset bit, 20-24
ECINIT[15:0] field, 7-117
ECOUNT[15:0] field, 7-116
EHR (enable host reads) bit, 8-25
EHW (enable host writes) bit, 8-25
elfloader.exe, 17-23
ELSI bit, 25-9, 25-42, 25-43, 25-44
EMISO bit, 22-21, 22-45
EMISO (enable MISO) bit, 22-45
emulation, and timer counter, 10-45
EMU_RUN bit, 10-43, 10-52
enable host reads (EHR) bit, 8-25
enable host writes (EHW) bit, 8-25
Enable Interrupts (STI) instruction, 3-79
enable MISO (EMISO) bit, 22-45
ENABLE_SUSPENDM (suspend mode output enable) bit, 26-99
enabling
general-purpose timers, 10-38
interrupts, 6-11
endian format
data and instruction storage, 3-69
end of cycle time for PIO access transfers (TEOC_REG_PIO) bits, 21-64
end of cycle time for register access transfers (T2_REG) bits, 21-63
end of data (DAT_END_MASK) bit, 27-67
END_ON_TERM (end/terminate select) bit, 21-49
endpoint number (EPNUM) bits, 26-145
endpoint x Rx enable (EPx_RX_ENA) bits, 26-98
endpoint x Tx enable (EPx_TX_ENA) bits, 26-98
end/terminate select (END_ON_TERM) bit, 21-49
EN (enable) bit, 15-81
ENICPLB, 3-9
EP0_NAK_LIMIT (EP0 NAK limit) bits, 26-130
EP0 NAK limit (EP0_NAK_LIMIT) bits, 26-130
EP0_RX_COUNT (received byte count in EP0 FIFO) bits, 26-128
EP bit, 31-46
EP halted after a NAK (NAK_TIMEOUT_H) bit, 26-113, 26-117
EPIF bit, 31-27, 31-50
EPIM bit, 31-27, 31-49
EPIS bit, 31-27, 31-49
EPNUM (endpoint number) bits, 26-145
EPPI
block diagram, 15-5
control signal polarities, 15-80
data width, 15-80
GP output, 15-12, 15-13
operating modes, 15-80
EPPI clipping (EPPIx_CLIP) register, 15-74, 15-98

EPPI_CONTROL (EPPI control register), 15-80
EPPI control register (EPPI_CONTROL), 15-80
EPPI vertical transfer count (EPPIx_VCOUNT) register, 15-74
EPPIx_CLIP (EPPI clipping) register, 15-74, 15-98
EPPIx_CLKDIV (EPPIx clock divide) register, 15-93
EPPIx clock divide (EPPIx_CLKDIV) register, 15-93
EPPIx_CONTROL register, 15-74, 15-82
EPPIx_FRAME (EPPIx lines per frame) register, 15-90
EPPIx_FS1P_AVPL (EPPIx FS1 period) register, 15-74, 15-97
EPPIx FS1 period (EPPIx_FS1P_AVPL) register, 15-74, 15-97
EPPIx_FS1W_HBL (EPPIx FS1 width) register, 15-94
EPPIx FS1 width (EPPIx_FS1W_HBL) register, 15-94
EPPIx FS2 period (EPPIx_FS2P_LAVF) register, 15-74, 15-97
EPPIx_FS2P_LAVF (EPPIx FS2 period) register, 15-74, 15-97
EPPIx FS2 width (EPPIx_FS2W_LVB) register, 15-74, 15-95
EPPIx_FS2W_LVB (EPPIx FS2 width) register, 15-74, 15-95
EPPIx_HCOUNT (EPPIx horizontal transfer count) register, 15-93
EPPIx_HDELAY (EPPIx horizontal delay count) register, 15-74, 15-92
EPPIx horizontal delay count (EPPIx_HDELAY) register, 15-74, 15-92
EPPIx horizontal transfer count (EPPIx_HCOUNT) register, 15-93
EPPIx_LINE (EPPIx samples per line) register, 15-74, 15-90
EPPIx lines per frame (EPPIx_FRAME) register, 15-90
EPPIx samples per line (EPPIx_LINE) register, 15-74, 15-90
EPPIx_STATUS (EPPI status) register, 15-80
EPPIx status (EPPIx_STATUS) register, 15-80
EPPIx_VCOUNT (EPPI vertical transfer count) register, 15-74, 15-92
EPPIx_VDELAY (EPPI vertical delay count) register, 15-91
EPPIx vertical delay count (EPPIx_VDELAY) register, 15-91
EPPIx vertical transfer count (EPPIx_VCOUNT) register, 15-92
EPS (even parity select) bit, 25-29
EPx_RX_ENA (endpoint x Rx enable) bits, 26-98
EPx_RX_E (USB Rx endpoint x interrupt enable) bits, 26-108
EPx_RX (USB Rx endpoint x interrupt) bits, 26-106
EPx_TX_ENA (endpoint x Tx enable) bits, 26-98
EPx_TX_E (USB Tx endpoint x interrupt enable) bits, 26-107
EPx_TX (USB Tx endpoint x interrupt) bits, 26-105
ERBFI bit, 25-9, 25-17, 25-41, 25-42, 25-43
ERR_DET (Error Detected in Preamble) bit, 21-49, 21-55, 21-56
ERR_DET (preamble error detected) bit, 15-80
ERR_NCOR (Error Not Corrected in Preamble) bit, 21-49, 21-55, 21-56

ERR_NCOR (preamble error not corrected) bit, 15-80
error
  bus exception, 29-4
  hardware interrupt, 29-4
Error Detected in Preamble (ERR_DET) bit, 21-49, 21-55, 21-56
error frames, CAN, 31-31
ERROR_H (timeout error) bit, 26-113
Error Not Corrected in Preamble (ERR_NCOR) bit, 21-49, 21-55, 21-56
error-passive interrupt, CAN, 31-27
ERROR_RH (timeout error indicator) bit, 26-123
errors
  DMA, 7-37
  misalignment of data, 3-76
  not detected by DMA hardware, 7-38
  startup, and timers, 10-11
error signals, SPI, 22-23 to 22-25
ERROR_TH (timeout error indicator) bit, 26-117
error type bit, 10-43
error warning receive interrupt, CAN, 31-27
error warning transmit interrupt, CAN, 31-27
ERR_TYP[1:0] field, 10-10, 10-42, 10-43, 10-52
ETBEI bit, 25-7, 25-16, 25-41, 25-42, 25-43
event counter, CAN, 31-28
event handling, 6-6
events
  default mapping, 6-15
  definition, 6-6
  types of, 6-6
event vector table (EVT), 6-2
EVT1 register, 17-8
EWLREC[7:0] field, 31-85
EWLTEC[7:0] field, 31-85
EWRIF bit, 31-27, 31-50
EWRIM bit, 31-27, 31-49
EWRIS bit, 31-27, 31-49
EWTIF bit, 31-27, 31-50
EWTIM bit, 31-27, 31-49
EWTIS bit, 31-27, 31-49
exclusive (definition), 3-80
EXT_CLK mode, 10-34 to 10-35, 10-47
  control bit and register usage, 10-51
  flow diagram, 10-35
External Access Bus (EAB), 2-4, 2-5, 2-24
External Bus (DEB), DMA, 2-5, 2-17
external bus interface unit, *See* EBIU
External Bus Interface Unit (EBIU), 2-2
External Bus Interface Unit (EBIU) Diagram, 5-4
external crystal, 1-30
External memory, 2-2
external memory, 3-50
  design issues, 19-5
external memory map
  figure, 5-3
EXTEST instruction, B-6
EXTID[15:0] field, 31-52, 31-56
EXTID[17:16] field, 31-51, 31-54

F

F1_ACT bits, 15-97
F1VB_AD bits, 15-95
F1VB_BD bits, 15-95
F2_ACT bits, 15-97
F2VB_AD bits, 15-95
F2VB_BD bits, 15-95
FAST bit, 23-32, 23-34
fast mode, TWI, 23-10
FCPOL (flow control pin polarity) bit, 25-32
FCZ0EN, 29-45

FCZ1EN, 29-45
FDF bit, 31-20, 31-51
FE (framing error) bit, 25-35, 25-36
FER bit, 31-85
FERREN, 29-45
FFE bit, 25-50, 25-51
Field (FLD) bit, 21-49, 21-55, 21-56
field (FLD) bit, 15-80
field select/trigger (FLD_SEL) bit, 15-81
FIFO_COUNT (FIFO count) bits, 27-68
FIFO count) (FIFO_COUNT) bits, 27-68
FIFOEMPTY bit, 8-27
FIFO Error (FERR) interrupt event, 29-36
FIFO Error interrupt enable, 29-45
FIFO_FLUSH (flush FIFOs) bit, 21-49
FIFO flush (HOST_FLUSH) bit, 8-25
FIFOFULL bit, 8-27
FIFO_FULL_R (detected FIFO not empty) bit, 26-123
FIFO_NOT_EMPTY_T (data packet in FIFO indicator) bit, 26-117
FIFO regular watermark (FIFO_RWM) bits, 15-82
FIFO_RWM (FIFO regular watermark) bits, 15-82
FIFO urgent watermarks (FIFO_UWM) bits, 15-82
FIFO_UWM (FIFO urgent watermarks) bits, 15-82
finish control command, DMA, 7-41
Fixed Pattern Matching select (FIXEDPM) bit, 29-66
Flash, 2-2
flash interface, 19-8
FLD (Field) bit, 21-49, 21-55, 21-56
FLD (field) bit, 15-80
FLD_SEL (field select/trigger) bit, 15-81
flex descriptors, 7-3
FLGx bit, 22-46
FLOW[2:0] field, 7-30, 7-31, 7-66, 7-80, 7-82
FLOW bit, 8-6
flow charts
  CAN receive operation, 31-19
  CAN transmit operation, 31-15
  DMA, 7-26, 7-27
  general-purpose timers interrupt structure, 10-9
  SPI core-driven, 22-39
  SPI DMA, 22-40
  timer EXT_CLK mode, 10-35
  timer PWM_OUT mode, 10-14
  timer WDTH_CAP mode, 10-26
  TWI master mode, 23-24
  TWI slave mode, 23-23
FLOW mode, DMA, 7-24
FLOW value, DMA, 7-28
FLSx bit, 22-12, 22-46
FLSx (slave select enable) bits, 22-46
flush endpoint FIFO (FLUSHFIFO) bit, 26-113
flush endpoint FIFO (FLUSHFIFO_R) bit, 26-123
flush endpoint FIFO (FLUSHFIFO_T) bit, 26-117
FLUSHFIFO (flush endpoint FIFO) bit, 26-113
FLUSHFIFO_R (flush endpoint FIFO) bit, 26-123
FLUSHFIFO_T (flush endpoint FIFO) bit, 26-117
FLUSH instruction, 3-41
FLUSHINV instruction, 3-41
FMD bit, 31-51
FORCE_DATATOGGLE_T (force endpoint data toggle) bit, 26-117
force endpoint data toggle (FORCE_DATATOGGLE_T) bit, 26-117

FORCE_MSEL (force PLL frequency multiplier) bits, 26-142
force PLL frequency multiplier (FORCE_MSEL) bits, 26-142
formatting enable (RGB_FMT_EN) bit, 15-82
FPE bit, 25-50, 25-51
Frame Counter 0 Zero (FCZ0) interrupt event, 29-34
Frame Counter 0 Zero interrupt enable, 29-45
Frame Counter 1 Zero (FCZ1) interrupt event, 29-34
Frame Counter 1 Zero interrupt enable, 29-45
framed serial transfers, characteristics, 24-34
framed/unframed data, 24-33
frame interrupt enable (FRM_INT_EN) bit, 28-41
frame interrupt status (FRM_INT_STAT) bit, 28-41
Frame Locked (FLOCK) bit, 29-25
Frame Locked to Unlocked (FL2U) interrupt event, 29-35
FRAME_NUMBER (USB frame number) bits, 26-111
frame sync
  active high/low, 24-35
  early, 24-38
  early/late, 24-38
  external/internal, 24-34
  internal, 24-28
  internally generated, 24-69
  late, 24-38
  multichannel mode, 24-20
  sampling edge, 24-35
  SPORT options, 24-33
frame sync configuration (FS_CFG) bits, 15-81
frame sync divider[15:0] field, 24-69
frame synchronization
  and SPORT, 24-3
frame sync pulse
  use of, 24-55
  when issued, 24-55
frame sync signal, control of, 24-54, 24-60
Frame Track Overflow Error (FTERR_OVR) bit, 21-49, 21-55, 21-56
frame track overflow (FTERR_OVR) bit, 15-80
Frame Track Underflow Error (FTERR_UNDR) bit, 21-49, 21-55, 21-56
frame track underflow (FTERR_UNDR) bit, 15-80
Frame Unlocked to Locked (FU2L) interrupt event, 29-35
FREQ[1:0] field, 18-18, 18-28
frequencies, clock and frame sync, 24-28
FRM_INT_EN (frame interrupt enable) bit, 28-41
FRM_INT_STAT (frame interrupt status) bit, 28-41
FS_CFG (frame sync configuration) bits, 15-81
FSDEV (full- or high-speed device indicator) bit, 26-134, 26-136
FSDR bit, 24-24, 24-71
FS_EOF1 (full-speed EOF 1) bits, 26-140
FTERR_OVR (frame track overflow) bit, 15-80
FTERR_OVR (Frame Track Overflow Error) bit, 21-49, 21-55, 21-56
FTERR_UNDR (frame track underflow) bit, 15-80
FTERR_UNDR (Frame Track Underflow Error) bit, 21-49, 21-55, 21-56

full duplex, 24-4, 24-8
SPI, 22-1
FULL_ON bit, 18-27
full on mode, 1-31, 18-8
full- or high-speed device indicator (FSDEV) bit, 26-134, 26-136
full-speed EOF 1 (FS_EOF1) bits, 26-140
FUNCTION_ADDRESS (USB peripheral device address) bits, 26-102

G

GAIN[1:0] field, 18-18, 18-28
gain levels, 18-18
GCALL bit, 23-31
GEN bit, 23-28
general call address, TWI, 23-10
general-purpose interrupts, 6-6, 6-7
general-purpose ports, 1-14, 9-1 to 9-74
general-purpose timers, 10-1 to 10-62
aborting, 10-24
and startup errors, 10-11
autobaud mode, 10-33
block diagram, 10-3
buffer registers, 10-47
capture mode, 10-7
clock source, 10-5
code examples, 10-53
control bit summary, 10-51
counter, 10-6
disable timing, 10-24
disabling, 10-39
enabling, 10-6, 10-35, 10-38
error detection, 10-10
EXT_CLK mode, 10-47
external interface, 10-4
features, 10-2
flow diagram for EXT_CLK mode, 10-35
generating maximum frequency, 10-17
illegal states, 10-10, 10-11
general-purpose timers *(continued)*
internal interface, 10-5
internal timer structure, 10-4
interrupts, 10-6, 10-7, 10-16, 10-30
interrupt setup, 10-55
interrupt structure, 10-9
measurement report, 10-27, 10-28, 10-29
non-overlapping clock pulses, 10-59
output pad disable, 10-15
overflow, 10-6
periodic interrupt requests, 10-56
port setup, 10-53
preventing errors in PWM_OUT mode, 10-48
programming model, 10-35
PULSE_HI toggle mode, 10-18
PWM mode, 10-7
PWM_OUT mode, 10-13 to 10-24, 10-47
registers, 10-37
signal generation, 10-54
single pulse generation, 10-15
size of register accesses, 10-37
stopping in PWM_OUT mode, 10-23
three timers with same period, 10-19
two timers with non-overlapping clocks, 10-19
waveform generation, 10-16
WDTH_CAP mode, 10-25, 10-47
WDTH_CAP mode configuration, 10-61
WDTH_CAP mode flow diagram, 10-26
get more data (GM) bit, 22-45
GIRQ bit, 31-48
glitch filtering, UART, 25-14
GLOBAL_ENA (USB enable) bit, 26-98
global interrupts, CAN, 31-25
glueless connection, 19-8

GM bit, 22-30, 22-45
GM (get more data) bit, 22-45
GPIO, 9-1 to 9-74, 29-3
  interrupt request, 6-41
  pins, 9-2
ground plane, 19-15
Group cast/Broadcast Transmission Status Encodings, 29-141
GU_MULT4 (multiply row by 4) bit, 28-43
GU_TRANS (transparent color - G/U) bits, 28-46

H

H.100, 24-24, 24-27
handshake MDMA, 7-16, 7-45
  interrupts, 7-49
handshake MDMA control registers (HMDMAx_CONTROL), 7-111, 7-113
handshake MDMA current block count registers (HMDMAx_BCOUNT), 7-115
handshake MDMA current edge count registers (HMDMAx_ECOUNT), 7-116
handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), 7-118
handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), 7-118
handshake MDMA edge count urgent registers (HMDMAx_ECURGENT), 7-117
handshake MDMA initial block count registers (HMDMAx_BCINIT), 7-114
handshake MDMA initial edge count registers (HMDMAx_ECINIT), 7-117
handshake MDMA initial edge count registers (HMDMAx_ECINIT), 7-117
handshake memory DMA, 7-3
hardware reset, 17-5, 17-6, 17-8
Harvard architecture, 3-5
HDONEENx, 29-47
header checksum field
  HDRCHK, 17-30
header signature
  HDRSGN, 17-31
hibernate state, 1-33, 18-11, 18-20
  and CAN, 31-40
HIGH_EVEN (upper limit for even bytes (luma) bits, 15-98
high frequency design considerations, 19-5
HIGH_ODD (upper limit for odd bytes (chroma) bits, 15-98
high- or full-speed device indicator (FSDEV) bit, 26-134, 26-136
high-speed EOF 1 (HS_EOF1) bits, 26-139
high speed mode enable (HS_ENABLE) bit, 26-99
high speed mode flag (HS_MODE) bit, 26-99
HIRQ (host interrupt request) bit, 8-27
HMDMA, 7-16
HMDMAEN bit, 7-45, 7-47, 7-113
HMDMAx_BCINIT (handshake MDMA configuration registers), 7-46, 7-114
HMDMAx_BCOUNT (handshake MDMA current block count registers), 7-46, 7-115
HMDMAx_CONTROL (handshake MDMA control registers), 7-8, 7-111, 7-113

HMDMAx_ECINIT (handshake MDMA initial edge count registers), 7-47, 7-117
HMDMAx_ECOUNT (handshake MDMA current edge count registers), 7-47, 7-116
HMDMAx_ECOVERFLOW (handshake MDMA edge count overflow interrupt registers), 7-118
HMDMAx_ECURGENT (handshake MDMA edge count urgent registers), 7-117
HMVIP, 24-27
HOST acknowledge mode timeout (HOST_TIMEOUT) register, 8-29
HOST_CONFIG (HOST configuration) word, 8-6
HOST configuration word (HOST_CONFIG), 8-6
HOST_CONTROL (HOST control) register, 8-25
HOST control (HOST_CONTROL) register, 8-25
host enable (HOST_EN) bit, 8-25
HOST_END (host endianess) bit, 8-25
host endianess (HOST_END) bit, 8-25
HOST_EN (host enable) bit, 8-25
HOST_FLUSH (FIFO flush) bit, 8-25
host handshake (HSHK) bit, 8-27
host interrupt request (HIRQ) bit, 8-27
host negotiation request (HOST_REQ) bit, 26-134, 26-136
host ready override (HRDY_OVR) bit, 8-25
HOST_REQ (host negotiation request) bit, 26-134, 26-136
HOST_STATUS (HOST status) register, 8-27
HOST status (HOST_STATUS) register, 8-27
host terminate current transfer interrupt mask (HOST_TERM_XFER_MASK) bit, 21-55
host terminate current transfer interrupt status (HOST_TERM_XFER_INT) bit, 21-57
host termination (ATAPI_HOST_TERM) bit, 21-60
HOST_TERM_XFER_INT (host terminate current transfer interrupt status) bit, 21-57
HOST_TERM_XFER_MASK (host terminate current transfer interrupt mask) bit, 21-55
host timeout count (COUNT_TIMEOUT) bits, 8-29
HOST_TIMEOUT (HOST acknowledge mode timeout) register, 8-29
host timeout (TIMEOUT) bit, 8-27
hours[3:0] field, 14-21, 14-23
hours[4] bit, 14-21, 14-23
hours event flag bit, 14-22
hours interrupt enable bit, 14-21
HRDY_OVR (host ready override) bit, 8-25
HS_ENABLE (high speed mode enable) bit, 26-99
HS_EOF1 (high-speed EOF 1) bits, 26-139
HSHK (host handshake) bit, 8-27
HS_MODE (high speed mode flag) bit, 26-99

I

I²C bus standard, 1-15, 23-2
I²S, 1-19
  format, 24-13
  serial devices, 24-2, 24-7

ICIE (illegal gray/binary code interrupt enable) bit, 13-28
ICII (illegal gray/binary code interrupt identifier) bit, 13-28
ICLKGEN (internal clock generation) bit, 15-81
ICPLB Address registers (ICPLB_ADDRx), 3-64
ICPLB_ADDRx (ICPLB Address registers), 3-64
ICPLB Data registers (ICPLB_DATAx), 3-59
ICPLB_DATAx (ICPLB Data registers), 3-59
ICPLB Fault Address register (ICPLB_FAULT_ADDR), 3-67
ICPLB_FAULT_ADDR (ICPLB Fault Address register), 3-67
ICPLB_STATUS (ICPLB Status register), 3-66
ICPLB Status register (ICPLB_STATUS), 3-66
IDE bit, 31-54
IDE interface, 21-1
idle state
  waking from, 6-13
IEEE 1149.1 standard, *See* JTAG standard
IFSGEN (internal frame sync generation) bit, 15-81
image data format (IMG_FORM) bit, 28-37
image FIFO status (IMG_STAT) bits, 28-37
IMC, 3-9
IMEM_CONTROL (Instruction Memory Control Register), 3-9
IMEM_CONTROL (Instruction Memory Control register), 3-9, 3-52
IMEM_CONTROL (instruction memory control register), 3-9
IMG_FORM (image data format) bit, 28-37
IMG_STAT (image FIFO status) bits, 28-37
INCOMPTX_RH (large packet split) bit, 26-123
INCOMPTX_R (large packet split) bit, 26-123
INCOMPTX_T (large packet split) bit, 26-117
increase PLL charge pump current (TM_SELC) bit, 26-142
index (definition), 3-80
INIT bit, 17-40
initcall address/symbol command, 17-41
initcode routines, 17-39
initializing
  CAN, 31-10
  DMA, 7-25
init initcode.dxe command, 17-41
inner loop address increment registers
  (DMAx_X_MODIFY), 7-97
  (MDMA_yy_X_MODIFY), 7-97
inner loop count registers
  (DMAx_X_COUNT), 7-92
  (MDMA_yy_X_COUNT), 7-92
INPDIS (CUD and CDZ input disable) bit, 13-27
input clock, *See* CLKIN
input delay bit, 18-26
Inserting Wait States using ARDY (figure), 5-77
instruction bit scan ordering, B-5
instruction cache
  coherency, 3-20
Instruction CPLB Enable, 3-9
instruction fetches, 3-52
Instruction Memory Control Register (IMEM_CONTROL), 3-9

Instruction Memory Control register (IMEM_CONTROL), 3-9, 3-52
instruction register, B-2, B-4
instructions, 1-34
  interlocked pipeline, 3-71
  load store, 3-71
  See also instructions by name
  stored in memory, 3-70
  synchronizing, 3-73
Instruction Test Command register (ITEST_COMMAND), 3-24
Instruction Test Data registers (ITEST_DATAx), 3-25
Instruction Test registers, 3-23 to 3-26
Integrated Drive Electronics interface, 21-1
INT_ENA (interrupt enable) bit, 26-145
Interface
  D Port, 2-9
  On-Chip L2 Memory, 2-11
interfaces
  internal memory, 5-7
  RTC, 14-3
interleaving
  of data in SPORT FIFO, 24-62
  SPORT data, 24-8
internal bank, G-12
internal boot ROM, 17-1
internal clock generation (ICLKGEN) bit, 15-81
internal clocks, 2-5
internal/external frame syncs, *See* frame sync
internal frame sync generation (IFSGEN) bit, 15-81
internal interfaces, 2-1
internal memory, 1-6, 3-5
  interfaces, 5-7
internal supply regulator, shutting off, 18-20
interrupt
  enabling and disabling, 3-79
  priority watermark, 3-40
interrupt channels, UART, 25-41
interrupt conditions, UART, 25-44
interrupt enable (INT_ENA) bit, 26-145
interrupt handler and DMA synchronization, 7-68
interrupt mask (CNT_IMASK) register, 13-25, 13-28, A-91, A-92
interrupt mode (INT_MODE) bit, 8-25
interrupt mode (INT_MODE) bit, 8-25
interrupt output, SPI, 22-25
Interrupt Priority register (IPRIO), 3-40
interrupt request enable bit, 10-43
interrupt request lines, peripheral, 6-2
interrupts, 6-1 to 6-43
  CAN, 31-24
  clearing requests, 6-41
  configuring and servicing, 19-2
  control of system, 6-6
  core timer, 11-3
  default mapping, 6-7
  definition, 6-6
  determining source, 6-12
  DMA channels, 6-13
  DMA_ERROR, 7-37
  DMA error, 7-88
  DMA overflow, 7-50
  DMA queue completion, 7-70
  enabling, 6-11
  general-purpose, 6-6, 6-7
  general-purpose timers, 10-6, 10-7, 10-16, 10-30
  generated by peripheral, 6-22
  handshake MDMA, 7-49
  initialization, 6-22
  inputs and outputs, 6-10
  mapping, 6-11
  mask function, 6-14

interrupts *(continued)*
multiple sources, 6-24
peripheral, 6-6, 6-10, 6-10 to 6-22
peripheral IDs, 6-15
peripheral interrupt events, 6-15
prioritization, 6-11
processing, 6-22
programming examples, 6-40 to 6-43
reset, 17-10
routing overview, 6-3, 6-4, 6-5
RTC, 14-16
shared, 6-11
software, 6-10
SPI, 22-26, 22-52
SPORT error, 24-41
SPORT RX, 24-41, 24-65
SPORT TX, 24-41, 24-62
to wake core from idle, 6-13
UART, 25-15
use in managing a descriptor queue, 7-67
interrupt service routine, determining source of interrupt, 6-12
interrupt status registers
(DMAx_IRQ_STATUS), 7-84, 7-85
(MDMA_yy_IRQ_STATUS), 7-84, 7-85
INT_MODE (interrupt mode) bit, 8-25
invalid cache line (definition), 3-80
I/O interface to peripheral serial device, 24-4
I/O memory space, 1-9
IORDY_EN (IORDY enable) bit, 21-49
IPRIO (Interrupt Priority register), 3-40
IRCLK bit, 24-57, 24-59
IrDA
receiver, 25-14
transmitter, 25-13
IrDA mode, 25-50
IREN bit, 25-50
IRFS bit, 24-34, 24-57, 24-60
IRPOL bit, 25-14
IRQ_ENA bit, 10-43, 10-51, 10-53
isochronous transfer enable (ISO_R) bit, 26-123
isochronous transfer enable (ISO_T) bit, 26-117
isochronous update enable (ISO_UPDATE) bit, 26-99
ISO_R (isochronous transfer enable) bit, 26-123
ISO_T (isochronous transfer enable) bit, 26-117
ISO_UPDATE (isochronous update enable) bit, 26-99
ISR and multiple interrupt sources, 6-24
ITCLK bit, 24-51, 24-53
ITEST_COMMAND (Instruction Test Command register), 3-24
ITEST_DATAx (Instruction Test Data registers), 3-25
ITEST register
ITEST_COMMAND, 3-23
ITEST_DATA0, 3-23
ITEST_DATA1, 3-23
ITEST registers, 3-23
ITFS bit, 24-21, 24-34, 24-51, 24-54
ITHR[15:0] field, 7-118
ITU interlaced/progressive (ITU_TYPE) bit, 15-81
ITU output with internal blanking (BLANKGEN) bit, 15-81
ITU_TYPE (ITU interlaced/progressive) bit, 15-81

J

JTAG, B-1, B-3, B-4

K

keypad
enable/disable, 30-4
input matrix programmability, 30-4
interface, 30-3
interface overview, 30-1
KPAD_CTL register, 30-4
KPAD_PRESCALE register, 30-9
operation, 30-2
programming examples, 30-20
programming model, 30-9
registers
interrupt generation when X-Key pressed, 30-17
KPAD_CTL, 30-10, A-36
KPAD_MSEL, 30-10, A-36
KPAD_PRESCALE, 30-10, A-36
KPAD_ROWCOL, 30-10, A-36
KPAD_SOFTEVAL, 30-10, A-36
KPAD_STAT, 30-10, A-36
state diagram, 30-8
keypad control (KPAD_CTL) register, 30-10, A-36
keypad enable (KPAD_EN) bit, 30-10
keypad interrupt status (KPAD_IRQ) bit, 30-18
keypad multiplier select (KPAD_MSEL) register, 30-10, 30-15, A-36
keypad row-column (KPAD_ROWCOL) register, 30-10, 30-15, A-36
keypad software evaluate (KPAD_SOFTEVAL) register, 30-20
key prescale (KPAD_PRESCALE) register, 30-10, 30-13, A-36
key prescale value (KPAD_PRESCALE_VAL) bits, 30-13
key press current status (KPAD_PRESSED) bit, 30-18
key software evaluate (KPAD_SOFTEVAL) register, 30-10, A-36
KPAD_COL (columns value pressed) bits, 30-15
KPAD_COLEN (column enable width) bits, 30-10
KPAD_CTL (keypad control) register, 30-10, A-36
KPAD_EN (keypad enable) bit, 30-10
KPAD_IRQ (keypad interrupt status) bit, 30-18
KPAD_IRQMODE (multikey press interrupt enable) bits, 30-10
KPAD_MROWCOL (multiple row/column keypress) bits, 30-18
KPAD_MSEL (keypad multiplier select) register, 30-10, 30-15, A-36
KPAD_PRESCALE (key prescale) register, 30-10, 30-13, A-36
KPAD_PRESCALE_VAL (key prescale value) bits, 30-13
KPAD_PRESSED (key press current status) bit, 30-18
KPAD_ROWCOL (keypad row-column) register, 30-10, 30-15, A-36
KPAD_ROWEN (row enable width) bits, 30-10
KPAD_ROW (rows value pressed) bits, 30-15
KPAD_SOFTEVAL_E (software programmable force evaluate) bit, 30-20
KPAD_SOFTEVAL (keypad software evaluate) register, 30-20
KPAD_SOFTEVAL (key software evaluate) register, 30-10, A-36
KPAD_STAT (keypad status) register, 30-10, 30-18, A-36

L

L1 data memory, 1-6
L1 Data Memory Architecture, 3-32
L1 Data SRAM, 3-30
L1 instruction memory, 1-6
  subbanks, 3-11
L1 Instruction Memory Bank Architecture, 3-14
L1 instruction memory Configuration, 3-9
L1 memory. *See* Level 1 (L1) memory; Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
L1 or L2 memory, DMA access stalls, 2-13
L1 scratchpad RAM, 1-6
L2 bus, 2-5
L2DMAPRIO, 17-106
L2 memory
  access bus arbitration, 2-13
  access latency and throughput, 2-13
  arbitration, 2-11
  interface, 2-11
  interface control logic clock rate, 2-11
L2 port
  access priority, 2-12
  access request, arbitration priority, 2-12
LARFS bit, 24-38, 24-57, 24-60
large descriptor mode, DMA, 7-22
large model mode, DMA, 7-83
large packet split (INCOMPTX_R) bit, 26-123
large packet split (INCOMPTX_RH) bit, 26-123
large packet split (INCOMPTX_T) bit, 26-117
late frame sync, 24-19, 24-38
Latency
  DMA data transfer, 2-13
  L2 memory access, 2-13
latency
  DMA, 7-32
LATFS bit, 24-38, 24-51, 24-55
least recently used algorithm (LRU), definition, 3-80
Level 1 (L1) Data Memory, 3-27
  configuration, 3-7
  sub-banks, 3-30
Level 1 (L1) Instruction Memory, 3-8
  architecture, 3-13
  configuration, 3-13
  DAG reference exception, 3-11
  dual-port capability, 3-11
  instruction cache, 3-13
  sub-bank organization, 3-8
Level 1 (L1) memory
  See also Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
  address alignment, 3-11
  architecture, 3-5
  definition, 3-80
  L1 Data SRAM, 3-7
  L1 Instruction SRAM, 3-6
  scratchpad data SRAM, 3-7
Level 2 (L2) memory, 3-47
  latency, 3-48, 3-49
  latency with cache off, 3-49
  latency with cache on, 3-48
  off-chip, 3-50
lines per frame) bits, 28-38
lines per frame (LPF) bit, 28-38
line terminations, SPORT, 24-10
Line Track Overflow Error (LTERR_OVR) bit, 21-49, 21-55, 21-56
line track overflow (LTERR_OVR) bit, 15-80
Line Track Underflow Error (LTERR_UNDR) bit, 21-49, 21-55, 21-56
line track underflow (LTERR_UNDR) bit, 15-80

little endian byte order, 23-49
little endian (definition), 3-80
load, speculative execution, 3-74
loader file, 17-23
load error indicator (DATAERROR_R) bit, 26-123
loader utility, 17-23
load operation, 3-70
load ordering, 3-72
LOCKCNT[15:0] field, 18-27
Lock Mechanism 0, 29-18
Lock Mechanism 1, 29-18
Lock Mechanism Select (LMECH) bit, 29-18
Logical Channel for Physical Channel x (LCHANPCx) field, 29-58
long response enable (CMD_L_RSP) bit, 27-58
LOOPBACK (loopback mode enable) bit, 25-32
loopback mode, UART, 25-32
LOSTARB bit, 23-36, 23-38
Lower PBS02 Half Page (PBS02L, Bits 63–0), 17-118
LOW_EVEN (lower limit for even bytes (luma) bits, 15-98
LOW_ODD (lower limit for odd bytes (chroma) bits, 15-98
low-speed device indicator (LSDEV) bit, 26-134, 26-136
low-speed EOF 1 (LS_EOF1) bits, 26-140
LRFS bit, 24-34, 24-35, 24-57, 24-60
LRU Priority Reset, 3-9
LRUPRIORST, 3-9
LSBF bit, 22-45
LSB first (LSBF) bit, 22-45
LSBF (LSB first) bit, 22-45
LSDEV (low-speed device indicator) bit, 26-134, 26-136
LS_EOF1 (low-speed EOF 1) bits, 26-140
LT_ERR_OVR bit, 30-11, 30-18, 30-20
LTERR_OVR (line track overflow) bit, 15-80
LTERR_OVR (Line Track Overflow Error) bit, 21-49, 21-55, 21-56
LTERR_UNDR (line track underflow) bit, 15-80
LTERR_UNDR (Line Track Underflow Error) bit, 21-49, 21-55, 21-56
LTFS bit, 24-21, 24-34, 24-35, 24-51, 24-55
luma FIFO error (YFIFO_ERR) bit, 15-80
Luma FIFO Overflow Error (YFIFO_ERR) bit, 21-49, 21-55, 21-56

M

MAA bit, 31-47
MAC
  pins, 30-6
MADDR[6:0] field, 23-35
mailboxes, CAN, 31-5
mailbox interrupts, CAN, 31-24
mapping
  default interrupt, 6-15
  peripheral to DMA, 6-14, 6-15
MASK_BUSYIRQ (mask not busy IRQ) bit, 20-22
mask not busy IRQ (MASK_BUSYIRQ) bit, 20-22
MASK_RDRDY (mask read data ready) bit, 20-22
mask read data ready (MASK_RDRDY) bit, 20-22
MASK_WBEDGE (mask write buffer edge detect) bit, 20-22
MASK_WBOVF (mask write buffer overflow) bit, 20-22
MASK_WRDONE (mask write done) bit, 20-22

mask write buffer edge detect (MASK_WBEDGE) bit, 20-22
mask write buffer overflow (MASK_WBOVF) bit, 20-22
mask write done (MASK_WRDONE) bit, 20-22
Master mode initialization, 29-112
master (MSTR) bit, 22-45
MAXCIE (max Count Interrupt Enable, 13-28
MAXCIE (max count interrupt enable) bit, 13-28
MAXCII (max count interrupt identifier) bit, 13-28
max count interrupt enable (MAXCIE) bit, 13-28
max count interrupt identifier (MAXCII) bit, 13-28
maximal count (CNT_MAX) register, 13-25, 13-32
Maximum Delay Register Updated interrupt enable, 29-45
Maximum Delay Register Updated (MDRU) interrupt event, 29-32
maximum individual packet size (MaxPktSize), 26-31, 26-32, 26-33, 26-34, 26-35, 26-36
Maximum Position Register Updated interrupt enable, 29-44
Maximum Position Register Updated (MPRU) interrupt event, 29-32
MAX_PACKET_SIZE_R (USB max Rx data in frame) bits, 26-122
MAX_PACKET_SIZE_T (USB max Tx data in frame) bits, 26-112
MaxPktSize (maximum individual packet size), 26-31, 26-32, 26-33, 26-34, 26-35, 26-36
MBCLK, 29-3
MBDI bit, 7-50, 7-113
MBIMn bit, 31-79, 31-80
MBPTR[4:0] field, 31-46
MBRIFn bit, 31-81, 31-82
MBRIRQ bit, 31-48
MBTIFn bit, 31-80, 31-81
MBTIRQ bit, 31-48
MCCRM[1:0] field, 24-71
MCDRXPE bit, 24-71
MCDTXPE bit, 24-71
MCMEN bit, 24-19, 24-71
MCOMP bit, 23-44, 23-46
MCOMPM bit, 23-43
MCx bit, 31-69
MDIR bit, 23-32, 23-34
MDMA controllers, 7-13
MDMA_ROUND_ROBIN_COUNT[4:0] field, 7-58, 7-120
MDMA_ROUND_ROBIN_PERIOD field, 7-57, 7-58, 7-120
MDMA_TFRCNT (MDMA transfer count) bits, 21-61
MDMA_XFER_ON (multi-word DMA transfer in progress) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
MDMA_yy_CONFIG (DMA configuration registers), 7-79
MDMA_yy_CURR_ADDR (current address registers), 7-90
MDMA_yy_CURR_DESC_PTR (current descriptor pointer registers), 7-108
MDMA_yy_CURR_X_COUNT (current inner loop count registers), 7-94, 7-95
MDMA_yy_CURR_Y_COUNT (current outer loop count registers), 7-101
MDMA_yy_IRQ_STATUS (interrupt status registers), 7-84, 7-85
MDMA_yy_NEXT_DESC_PTR (next descriptor pointer registers), 7-106

MDMA_yy_PERIPHERAL_MAP (peripheral map registers), 7-77
MDMA_yy_START_ADDR (start address registers), 7-88
MDMA_yy_X_COUNT (inner loop count registers), 7-92
MDMA_yy_X_MODIFY (inner loop address increment registers), 7-97
MDMA_yy_Y_COUNT (outer loop count registers), 7-99
MDMA_yy_Y_MODIFY (outer loop address increment registers), 7-103
MDn bit, 31-70
MDRUEN, 29-45
Meaning of CM Allocate Status, 29-151
Meaning of Transmission Status, 29-141
measurement report, general-purpose timers, 10-27, 10-28, 10-29
Media Oriented System Transport, 29-1
Media Transceiver module, 29-1
Memory
  access, latency and throughput, 2-13
  external, 2-2
  interface, 2-11
memory
  See also cache; Level 1 (L1) memory; Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory; Level 2 (L2) memory
  architecture, 1-5
  asynchronous interface, 19-8
  asynchronous region, 5-2
  configurations, 1-5
  external, 3-50
  how instructions are stored, 3-70
  internal, 1-6
  internal bank, G-12
  I/O space, 1-9
  L1 data, 1-6, 3-27
  L1 Data SRAM, 3-30
memory *(continued)*
  L1 instruction, 1-6
  L1 scratchpad, 1-6
  Level 2 (L2), 3-47
  management, 3-51
  moving data between SPORT and, 24-40
  on-chip, 1-6
  Page Descriptor Table, 3-54
  protection and properties, 3-51
  structure, 1-5
  terminology, 3-79
  transaction model, 3-69
Memory Architecture, 3-2
memory architecture, 3-2
memory conflict, DMA, 7-59
memory DMA, 7-13
  bandwidth, 7-53
  buffers, 7-14
  channels, 7-13
  descriptor structures, 7-72
  handshake operation, 7-16
  timing, 7-54
  transfer operation, starting, 7-14
  transfer performance, 2-23
  transfers, 7-3, 7-9
  word size, 7-14
Memory Management Unit (MMU), 3-51
Memory Map, 3-4
memory map, external (figure), 5-3
memory-mapped registers, *See* MMRs
memory-mapped registers (MMRs), 3-78, 3-79
memory page, 3-53
  attributes, 3-53
MEN bit, 23-32, 23-34
MERR bit, 23-44, 23-46
MERRM bit, 23-43
MFD[3:0] field, 24-23, 24-71
MFLOW field, 29-69

MFS, 29-3
MH2LEN, 29-45
MII
  pins, 30-6
MINCIE (min count interrupt enable) bit, 13-28
MINCII (min count interrupt identifier) bit, 13-28
min count interrupt enable (MINCIE) bit, 13-28
min count interrupt identifier (MINCII) bit, 13-28
minimal count (CNT_MIN) register, 13-25, 13-32
minutes[5:0] field, 14-21, 14-23
minutes event flag bit, 14-22
minutes interrupt enable bit, 14-21
MISO pin, 22-5, 22-6, 22-17, 22-19, 22-20, 22-22, 22-30
ML2HEN, 29-45
µ-law companding, 24-26, 24-31
MLF, 29-3
MLF analog pin, 29-2
MMCLK, 29-3
MMR
  offset, 29-4
MMR Port, 2-4
MMRs, 1-9
  address range, A-3
  width, A-3
MODE (DMA mode 0/1 selection) bit, 26-145
mode fault error, 22-24, 22-26
mode fault error (MODF) bit, 22-48
modes
  broadcast, 22-12, 22-19, 22-20
  multichannel, 24-17
  serial port, 24-12
  SPI master, 22-20, 22-26
  SPI slave, 22-20, 22-29
modes *(continued)*
  UART DMA, 25-24
  UART non-DMA, 25-22
MODF bit, 22-24, 22-48
MODF (mode fault error) bit, 22-48
modified (definition), 3-80
MOSI pin, 22-5, 22-6, 22-17, 22-19, 22-20, 22-22, 22-30
MOST®, 29-1
MOST® NetInterface, 29-1
moving data, serial port, 24-40
MPIVDD, 29-4
M (PLL multiplier select) bits, 26-142
MPROG bit, 23-36, 23-39
MPRUEN, 29-44
MRB bit, 31-47
MRTS (manual request to send) bit, 25-32
MRX, 29-2, 29-3
MRX input pin, 29-20
MRXONB High to Low interrupt enable, 29-45
MRXONB High to Low (MH2L) interrupt event, 29-34
MRXONB Low to High interrupt enable, 29-45
MRXONB Low to High (ML2H) interrupt event, 29-34
MSEL[5:0] field, 18-3, 18-4, 18-26
MSTR bit, 22-21, 22-45
MSTR (master) bit, 22-45
MTX, 29-3
MTXON, 29-3
multichannel frame, 24-22
multichannel frame delay field, 24-23
multichannel mode, 24-17
  enable/disable, 24-19
  frame syncs, 24-20
  SPORT, 24-20
multichannel operation, SPORT, 24-17 to 24-27

multi-DMA transfer done interrupt mask (MULTI_DONE_MASK) bit, 21-55
multi-DMA transfer done interrupt status (MULTI_DONE_INT) bit, 21-57
MULTI_DONE_INT (multi-DMA transfer done interrupt status) bit, 21-57
MULTI_DONE_MASK (multi-DMA transfer done interrupt mask) bit, 21-55
multikey press interrupt enable (KPAD_IRQMODE) bits, 30-10
multiple interrupt sources, 6-24
multiple row/column keypress (KPAD_MROWCOL) bits, 30-18
multiple slave SPI systems, 22-12
multiplexed with GPIO, 29-2
multiply row by 4 (BV_MULT4) bit, 28-44
multiply row by 4 (GU_MULT4) bit, 28-43
multiply row by 4 (RY_MULT4) bit, 28-42
MULTI_START (start multi-DMA Op) bit, 21-49
MULTI_TERM_INT (device terminate multi-DMA transfer interrupt status) bit, 21-57
MULTI_TERM_MASK (device terminate multi-DMA transfer interrupt mask) bit, 21-55
multi-word DMA transfer in progress (MDMA_XFER_ON) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
multi-word DMA transfer in progress (MULTI_XFER_ON) bit, 21-51
MULTI_XFER_ON (multi-word DMA transfer in progress) bit, 21-51
MUXy (port x bit y) bits, 9-47
MVIP-90, 24-27
MWE (SDIO interrupt moving window enable) bit, 27-72
MXEGND, 29-4
MXI, 29-3
MXO, 29-3
MXVR, 29-1
MXVR_AADDR, 29-55
MXVR_AADDR (MXVR alternate address) register, 29-55
MXVR allocation table (MXVR_ALLOC_x) registers, 29-56
MXVR Allocation Table Registers, 29-55
MXVR_ALLOC_x, 29-55
MXVR_ALLOC_x (MXVR allocation table) registers, 29-56
MXVR alternate address (MXVR_AADDR) register, 29-55
MXVR Alternate Address Register, 29-55
MXVR_AP_CTL, 29-75
MXVR_AP_CTL (MXVR asynchronous packet control) register, 29-75
MXVR_AP_CTL register, 29-75
MXVR_APRB_CURR_ADDR (MXVR asynchronous packet receive buffer current address) register, 29-78
MXVR_APRB_CURR_ADDR register, 29-78
MXVR_APRB_START_ADDR (MXVR asynchronous packet receive buffer start address) register, 29-78
MXVR_APRB_START_ADDR register, 29-77
MXVR_APTB_CURR_ADDR (MXVR asynchronous packet transmit buffer current address) register, 29-80
MXVR_APTB_CURR_ADDR register, 29-79

MXVR_APTB_START_ADDR (MXVR asynchronous packet transmit buffer start address) register, 29-79
MXVR_APTB_START_ADDR registers, 29-79
MXVR asynchronous packet control (MXVR_AP_CTL) register, 29-75
MXVR Asynchronous Packet Control Register, 29-75
MXVR asynchronous packet receive buffer current address (MXVR_APRB_CURR_ADDR) register, 29-78
MXVR Asynchronous Packet Receive Buffer Current Address Register, 29-78
MXVR asynchronous packet receive buffer start address (MXVR_APRB_START_ADDR) register, 29-78
MXVR Asynchronous Packet Receive Buffer Start Address Register, 29-77
MXVR asynchronous packet transmit buffer current address (MXVR_APTB_CURR_ADDR) register, 29-80
MXVR Asynchronous Packet Transmit Buffer Current Address Register, 29-79
MXVR asynchronous packet transmit buffer start address (MXVR_APTB_START_ADDR) register, 29-79
MXVR Asynchronous Packet Transmit Buffer Start Address Register, 29-79
MXVR Bit Clock, 29-3
MXVR_BLOCK_CNT, 29-94
MXVR_BLOCK_CNT (MXVR block counter) register, 29-94
MXVR block counter (MXVR_BLOCK_CNT) register, 29-94
MXVR Block Counter Register, 29-94
MXVR_CM_CTL (MXVR control message control) register, 29-81
MXVR_CM_CTL register, 29-80
MXVR_CMRB_CURR_ADDR, 29-83
MXVR_CMRB_CURR_ADDR (MXVR control message receive buffer current address) register, 29-84
MXVR_CMRB_START_ADDR, 29-82
MXVR_CMRB_START_ADDR (MXVR control message receive buffer start address) register, 29-83
MXVR_CMTB_CURR_ADDR, 29-85
MXVR_CMTB_CURR_ADDR (MXVR control message transmit buffer current address) register, 29-85
MXVR_CMTB_START_ADDR, 29-84
MXVR_CMTB_START_ADDR (MXVR control message transmit buffer start address) register, 29-85
MXVR_CONFIG, 29-13
MXVR_CONFIG (MXVR configuration) register, 29-13
MXVR configuration (MXVR_CONFIG) register, 29-13
MXVR Configuration Register, 29-13
MXVR control message control (MXVR_CM_CTL) register, 29-81
MXVR Control Message Control Register, 29-80
MXVR control message receive buffer current address (MXVR_CMRB_CURR_ADDR) register, 29-84
MXVR Control Message Receive Buffer Current Address Register, 29-83

MXVR control message receive buffer start address (MXVR_CMRB_START_ADDR) register, 29-83
MXVR Control Message Receive Buffer Start Address Register, 29-82
MXVR Control Message Transmit Buffer (CMTB), 29-138
MXVR control message transmit buffer current address (MXVR_CMTB_CURR_ADDR) register, 29-85
MXVR Control Message Transmit Buffer Current Address Register, 29-85
MXVR control message transmit buffer start address (MXVR_CMTB_START_ADDR) register, 29-85
MXVR Control Message Transmit Buffer Start Address Register, 29-84
MXVR Crystal Input, 29-3
MXVR Crystal Output, 29-3
MXVR_DELAY, 29-50
MXVR_DELAY (MXVR node frame delay) register, 29-50
MXVR DMA Channel x Current Address Registers, 29-71
MXVR DMA Channel x Current Transfer Count Registers, 29-74
MXVR DMA Channel x Start Address Registers, 29-69
MXVR DMA Channel x Transfer Count Registers, 29-72
MXVR_DMAx_CONFIG (MXVR DMAx data configuration) registers, 29-68
MXVR DMAx Configuration Registers, 29-59
MXVR_DMAx_COUNT, 29-72
MXVR_DMAx_CURR_ADDR, 29-71
MXVR_DMAx_CURR_ADDR (MXVR sync data DMAx current address) registers, 29-72
MXVR_DMAx_CURR_COUNT, 29-74
MXVR_DMAx_CURR_COUNT (MXVR sync data DMAx current loop count) registers, 29-74
MXVR_DMAx_START_ADDR, 29-69, 29-71
MXVR_DMAx_START_ADDR (MXVR sync data DMAx start address) registers, 29-69, 29-71
MXVR Enable (MXVREN) bit, 29-13
MXVR_FRAME_CNT_x (MXVR frame counter) registers, 29-90
MXVR_FRAME_CNT_x register, 29-90
MXVR frame counter (MXVR_FRAME_CNT_x) registers, 29-90
MXVR Frame Counter Registers, 29-90
MXVR Frame Sync, 29-3
MXVR_GADDR, 29-54
MXVR_GADDR (MXVR group address) register, 29-54
MXVR group address (MXVR_GADDR) register, 29-54
MXVR Group Address Register, 29-54
MXVR_INT_EN_0, 29-43
MXVR_INT_EN_0 (MXVR interrupt enable) register 0, 29-44
MXVR_INT_EN_1, 29-46
MXVR_INT_EN_1 (MXVR interrupt enable) register 1, 29-47
MXVR interrupt enable (MXVR_INT_EN_0) register 0, 29-44
MXVR interrupt enable (MXVR_INT_EN_1) register 1, 29-47
MXVR Interrupt Enable Register 0, 29-43

MXVR Interrupt Enable Register 1, 29-46
MXVR interrupt status (MXVR_INT_STAT_0) register 0, 29-30
MXVR interrupt status (MXVR_INT_STAT_1) register 1, 29-40
MXVR Interrupt Status Register 0, 29-29
MXVR Interrupt Status Register_1, 29-40
MXVR_INT_STAT_0, 29-29
MXVR_INT_STAT_0 (MXVR interrupt status) register 0, 29-30
MXVR_INT_STAT_1, 29-40
MXVR_INT_STAT_1 (MXVR interrupt status) register 1, 29-40
MXVR_LADDR, 29-53
MXVR_LADDR (MXVR logical address) register, 29-53
MXVR logical address (MXVR_LADDR) register, 29-53
MXVR Logical Address Register, 29-53
MXVR Master Clock, 29-3
MXVR Master Mode/Slave Mode Select (MMSM) bit, 29-14
MXVR_MAX_DELAY, 29-52
MXVR_MAX_DELAY (MXVR maximum node frame delay) register, 29-52
MXVR maximum node frame delay (MXVR_MAX_DELAY) register, 29-52
MXVR Maximum Node Frame Delay Register, 29-52
MXVR maximum node position (MXVR_MAX_POSITION) register, 29-49
MXVR Maximum Node Position Register, 29-49
MXVR_MAX_POSITION, 29-49
MXVR_MAX_POSITION (MXVR maximum node position) register, 29-49
MXVR Memory Map, 29-4, A-24
MXVR MMR address offsets, 29-4
MXVR node frame delay (MXVR_DELAY) register, 29-50
MXVR Node Frame Delay Register, 29-50
MXVR node position (MXVR_POSITION) register, 29-48
MXVR Node Position Register, 29-48
MXVR Parity select (PARITY) bit, 29-17
MXVR_PAT_DATA_x (MXVR pattern data) registers, 29-88
MXVR_PAT_EN_x (MXVR pattern enable) registers, 29-89
MXVR_PAT_EN_x registers, 29-88
MXVR pattern data (MXVR_PAT_DATA_x) registers, 29-88
MXVR Pattern Data Registers, 29-87
MXVR pattern enable (MXVR_PAT_EN_x) registers, 29-89
MXVR Pattern Enable Register, 29-88
MXVR Pattern Matching, 29-87
MXVR PHY Receiver Receiving Light, 29-3
MXVR_PLL_CTL_0 register, Initialization, 29-112
MXVR_POSITION, 29-48
MXVR_POSITION (MXVR node position) register, 29-48
MXVR Power Supply Pins, 29-3
MXVR Power-Up PHY Transmitter, 29-3
MXVR Power-Up PHY Transmitter (MTXON) bit, 29-16
MXVR Receive Data, 29-3
MXVR Register
  Phase Lock Loop Control, A-35

MXVR registers
list of, 29-4
MXVR remote read buffer current address (MXVR_RRDB_CURR_ADDR) register, 29-87
MXVR Remote Read Buffer Current Address Register, 29-86
MXVR remote read buffer start address (MXVR_RRDB_START_ADDR) register, 29-86
MXVR Remote Read Buffer Start Address Register, 29-86
MXVR_ROUTING_0 (MXVR routing 0) register, 29-91
MXVR routing 0 (MXVR_ROUTING_0) register, 29-91
MXVR Routing Registers, 29-91
MXVR_ROUTING_x register, Initialization, 29-113
MXVR_ROUTING_x registers, 29-91
MXVR_RRDB_CURR_ADDR, 29-86
MXVR_RRDB_CURR_ADDR (MXVR remote read buffer current address) register, 29-87
MXVR_RRDB_START_ADDR, 29-86
MXVR_RRDB_START_ADDR (MXVR remote read buffer start address) register, 29-86
MXVR Signal Pins, 29-3
MXVR signal pins, 29-2
MXVR_STATE_0, 29-19
MXVR State Registers, 29-19
MXVR_STATE_x state registers, 29-19
MXVR sync data DMAx current address (MXVR_DMAx_CURR_ADDR) registers, 29-72
MXVR sync data DMAx current loop count (MXVR_DMAx_CURR_COUNT) registers, 29-74
MXVR sync data DMAx start address (MXVR_DMAx_START_ADDR) registers, 29-69, 29-71
MXVR synchronous logical channel assignment (MXVR_SYNC_LCHAN_x) registers, 29-58
MXVR Synchronous Logical Channel Assignment Registers, 29-57
MXVR_SYNC_LCHAN_x, 29-57
MXVR_SYNC_LCHAN_x (MXVR synchronous logical channel assignment) registers, 29-58
MXVR Transmit Data, 29-3
MXVR Transmit Data Pin Enable (MTXEN) bit, 29-16

N

NA2IEN, 29-44
NAK bit, 23-28
NAK_TIMEOUT_H (EP halted after a NAK) bit, 26-113, 26-117
NAND address (NFC_ADDR) register, 20-18, 20-26, A-90
NAND command (NFC_CMD) register, 20-18, 20-27, A-90
NAND control (NFC_CTL) register, 20-18, 20-19, A-90
NAND data read (NFC_DATA_RD) register, 20-18, 20-28, A-90
NAND data width (NWIDTH) bit, 20-19
NAND data write (NFC_DATA_WR) register, 20-18, 20-28, A-90
NAND ECC count (NFC_COUNT) register, 20-18, 20-24, A-90
NAND ECC reset (NFC_RST) register, 20-18, 20-24, A-90

NAND flash controller
additional operations, 20-10
NFC accesses, 20-6
NFC error detection, 20-11
NFC external interface, 20-4
NFC interface block diagram, 20-4
overview, 20-1
page read, 20-9
page write, 20-8
NAND interrupt mask (NFC_IRQMASK) register, 20-18, 20-22, A-90
NAND interrupt status (NFC_IRQSTAT) register, 20-18, 20-21, A-90
NAND page control (NFC_PGCTL) register, 20-18, 20-25, A-90
NAND read data (NFC_READ) register, 20-18, 20-26, A-90
NAND status (NFC_STAT) register, 20-18, 20-20, A-90
NBUSYIRQ (not busy IRQ) bit, 20-21
NBUSY (not busy) bit, 20-20
NDPH bit, 7-28
NDPL bit, 7-28
NDSIZE[3:0] field, 7-23, 7-80, 7-83
legal values, 7-39
NetServices Layer 1, 29-1
Network Active (NACT) bit, 29-20
Network Active to Inactive interrupt enable, 29-44
Network Active to Inactive (NA2I) interrupt event, 29-31
Network Activity Detection, 29-109
Network Activity (NACT) bit, 29-31
Network Inactive to Active interrupt enable, 29-44
Network Inactive to Active (NI2A) interrupt event, 29-31
Network Initialization, 29-120
Network Lock, 29-117
network slave, 29-14
network timing master, 29-14
next descriptor pointer registers
(DMAx_NEXT_DESC_PTR), 7-106
(MDMA_yy_NEXT_DESC_PTR), 7-106
NFC
features, 20-2
NFC_ADDR (NAND address) register, 20-18, 20-26, A-90
NFC_CMD (NAND command) register, 20-18, 20-27, A-90
NFC_COUNT (NAND ECC count) register, 20-18, 20-24, A-90
NFC_CTL (NAND control) register, 20-18, 20-19, A-90
NFC_DATA_RD (NAND data read) register, 20-18, 20-28, A-90
NFC_DATA_WR (NAND data write) register, 20-18, 20-28, A-90
NFC_ECC0 (NAND ECC) register 0, 20-18, 20-23, A-90
NFC_ECC1 (NAND ECC) register 1, 20-18, 20-23, A-90
NFC_ECC1 (NAND ECC) register 2, 20-23
NFC_ECC2 (NAND ECC) register 2, 20-18, A-90
NFC_ECC3 (NAND ECC) register 3, 20-18, 20-23, A-90
NFC_ECCx (NAND ECC) registers, 20-18, 20-23, A-90
NFC_IRQMASK (NAND interrupt mask) register, 20-18, 20-22, A-90
NFC_IRQSTAT (NAND interrupt status) register, 20-18, 20-21, A-90
NFC_PGCTL (NAND page control) register, 20-18, 20-25, A-90
NFC_READ (NAND read data) register, 20-18, 20-26, A-90

NFC_RST (NAND ECC reset) register, 20-18, 20-24, A-90
NFC_STAT (NAND status) register, 20-18, 20-20, A-90
NI2AEN, 29-44
Node Initialization, 29-111
nominal bit rate, CAN, 31-12
nominal bit time, CAN, 31-11
NOPREBOOT, 16-58, 16-59
Normal Control Message Receive Enable (NCMRXEN) bit, 29-16
Normal Control Message Transmission, 29-142
Normal Control Message Transmit Buffer Entry Field Offsets, 29-143
normal frame sync mode, 24-38
normal timing, serial port, 24-38
not busy IRQ (NBUSYIRQ) bit, 20-21
no TxPktRdy for IN token (OVERRUN_R) bit, 26-123
no TxPktRdy for IN token (UNDERRUN_T) bit, 26-117
number of bytes to transfer (DATA_LENGTH) bits, 27-61
NWIDTH (NAND data width) bit, 20-19

O

OE (overrun error) bit, 25-35
offsets, DMA descriptor elements, 7-23
OI bit, 7-113
OIE bit, 7-113
onboard regulation, bypassing, 18-18
on-chip memory, 1-6
on-chip switching regulator controller, 18-16
one/two DMA channel modes (DMACFG) bit, 15-82
open drain drivers, 22-1
open drain outputs, 22-20
open page, G-1
operating modes, 18-7
  active, 1-31, 18-9
  deep sleep, 1-32, 18-10
  full on, 1-31, 18-8
  hibernate state, 1-33, 18-11
  sleep, 1-32, 18-9
  transition, 18-11, 18-12
operating mode (XFR_TYPE) bits, 15-81
OPSSn bit, 31-73
optical Phy, 29-1
optimization, of DMA performance, 7-50
ordering
  loads and stores, 3-72
  weak and strong, 3-72
oscilloscope probes, 19-18
OTP_ALTERNATE_HWAIT, 17-111
OTP_EBIU_AMBCTL, 17-115
OTP_EBIU_AMG, 17-114
OTP_EBIU_DEVCFG, 17-114
OTP_EBIU_DEVSEQ, 17-114
OTP_EBIU_FCTL, 17-115
OTP_EBIU_MODE, 17-114
OTP_INVALID, 17-111
OTP_LOAD_PBS00H, 17-111
OTP_LOAD_PBS01H, 17-111
OTP_LOAD_PBS01L, 17-111
OTP_LOAD_PBS02H, 17-111
OTP_LOAD_PBS02L, 17-111
OTP_LOAD_PBS03H, 17-111
OTP_LOAD_PBS03L, 17-111
OTP memory
  BFROM_OTP_COMMAND, 4-14
  bfrom_OtpCommand (), 4-8
  BFROM_OTP_READ, 4-17
  bfrom_OtpRead(), 4-8
  BFROM_OTP_WRITE, 4-18
  bfrom_OtpWrite(), 4-8
  error correction, 4-6
  map, 4-2
  overview, 4-1

OTP_NFC_CTL, 17-116
OTP_PLL_CTL, 17-113
OTP_PLL_DIV, 17-112
OTP_RESETOUT_HWAIT, 17-111
OTP_SET_PLL, 17-111
OTP_SET_VR, 17-111
OTP_SPI_BAUD, 17-112
OTP_SPI_FASTREAD, 17-112
OTP_START_PAGE, 17-116
OTP_TWI_CLKDIV, 17-112
OTP_TWI_PRESCALE, 17-112
OTP_TWI_TYPE, 17-111
OTP_VR_CTL, 17-113
OUT_DIS bit, 10-43, 10-51
outer loop address increment registers
  (DMAx_Y_MODIFY), 7-103
  (MDMA_yy_Y_MODIFY), 7-103
outer loop count registers
  (DMAx_Y_COUNT), 7-99
  (MDMA_yy_Y_COUNT), 7-99
OUT_FORM (output data format) bit, 28-37
output data format (OUT_FORM) bit, 28-37
output delay bit, 18-26
output pad disable, timer, 10-15
output pad disable bit, 10-43
outputs, programmable pins, 19-17
overflow interrupt, DMA, 7-50
overlay A enable (OVR_A_EN) bit, 28-37
overlay A horizontal end (A_HEND) bits, 28-39
overlay A horizontal start (A_HSTART) bits, 28-39
overlay A transparency (A_TRANSP) bits, 28-41
overlay A vertical end (A_VEND) bits, 28-40
overlay A vertical start (A_VSTART) bits, 28-40
overlay B enable (OVR_B_EN) bit, 28-37
overlay B horizontal end (B_HEND) bits, 28-39
overlay B horizontal start (B_HSTART) bits, 28-39
overlay B transparency (B_TRANSP) bits, 28-41
overlay B vertical end (B_VEND) bits, 28-40
overlay B vertical start (B_VSTART) bits, 28-40
overlay data format (OVR_FORM) bit, 28-37
overlay FIFO status (OVR_STAT) bits, 28-37
overlay interrupt enable (OVR_INT_EN) bit, 28-41
overlay interrupt status (OVR_INT_STAT) bit, 28-41
OVERRUN_R (no TxPktRdy for IN token) bit, 26-123
OVR_A_EN (overlay A enable) bit, 28-37
OVR_B_EN (overlay B enable) bit, 28-37
OVR_FORM (overlay data format) bit, 28-37
OVR_INT_EN (overlay interrupt enable) bit, 28-41
OVR_INT_STAT (overlay interrupt status) bit, 28-41
OVR_STAT (overlay FIFO status) bits, 28-37

P

PAB
  clocking, 18-1
  errors generated by SPORT, 24-41
PAB, Peripheral Access Bus, 2-4, 2-5, 2-15
PACKEN (pack/unpack enable) bit, 15-82
packet transaction status (STATUSPKT_H) bit, 26-113

packing, serial port, 24-26
pack/unpack enable (PACKEN) bit, 15-82
page 0x14, 17-114
page read pending (PG_RD_STAT) bit, 20-20
page read start (PG_RD_START) bit, 20-25
page write done (WR_DONE) bit, 20-21
page write pending (PG_WR_STAT) bit, 20-20
page write start (PG_WR_START) bit, 20-25
parity calculation result0 (ECC0) bits, 20-23
parity calculation result1 (ECC1) bits, 20-23
parity calculation result2 (ECC2) bits, 20-23
parity calculation result3 (ECC3) bits, 20-23
Parity Error interrupt enable, 29-45
Parity Error (PERR) interrupt event, 29-33
Pattern 0 Registers (PR0), 29-87
Pattern 1 Registers (PR1), 29-87
PBS00H, Bits 63–32(Upper PBS00 Half Page), 17-114
PBS01H, Bits 15–0(Upper PBS01 Half Page), 17-117
PBS01H, Bits 63–16(Upper PBS01 Half Page), 17-116
PBS02L, Bits 63–0(Lower PBS02 Half Page), 17-118
PD_SDDAT3 (SDH_DATA3 pull-down enable) bit, 27-72
PDWN bit, 18-26
pend enable (CMD_PEND_E) bit, 27-58
PEN (parity enable) bit, 25-29
PE (parity error) bit, 25-35, 25-36
Performance
  DAB, DCB, and DEB, 2-22
  DAB bus, 2-22
  DCB bus, 2-22
  DEB bus, 2-24
performance
  DMA, 7-52
  memory DMA, 7-53
  memory DMA transfers, 2-23
  optimization, DMA, 7-50
PERIOD_CNT bit, 10-13, 10-22, 10-27, 10-43, 10-51
Period count bit, 10-43
period value[15:0] field, 11-6
period value[31:16] field, 11-6
Peripheral
  DMA, 2-18
Peripheral Access Bus (PAB), 2-4, 2-5, 2-15
peripheral DMA, 7-10
peripheral DMA channels, 7-51
peripheral DMA start address registers, 7-88
peripheral DMA transfers, 7-2
peripheral error interrupts, 7-87
Peripheral ID (SDH_PIDx) bit, 27-73
peripheral interrupt request lines, 6-2
peripheral interrupts, 6-6, 6-10, 6-10 to 6-22
peripheral map registers
  (DMAx_PERIPHERAL_MAP), 7-77
  (MDMA_yy_PERIPHERAL_MAP), 7-77
peripherals, 1-2
  and buses, 1-3
  configuring for an IVG priority, 6-25
  default mapping to DMA, 7-10
  and DMA controller, 7-39
  DMA support, 1-3
  interrupt events, 6-15
  interrupt generated by, 6-22

peripherals *(continued)*
interrupt IDs, 6-15
interrupts, clearing, 6-41
level-sensitivity of interrupts, 6-42
list of, 1-2
mapping to DMA, 6-14, 6-15
remapping DMA assignment, 7-11
switching from DMA to non-DMA, 7-88
used to wake from idle, 6-13
PERREN, 29-45
PFx pin, 22-10
PG_RD_START (page read start) bit, 20-25
PG_RD_STAT (page read pending) bit, 20-20
PG_SIZE (page size) bit, 20-19
PG_WR_START (page write start) bit, 20-25
PG_WR_STAT (page write pending) bit, 20-20
Phy Receiver, 29-110
Phy Transmitter, 29-16
PHYWE bit, 18-28
pin assignment (PINTx_ASSIGN) registers, 9-63
pin interrupt edge clear (PINTx_EDGE_CLEAR) registers, 9-59
pin interrupt edge set (PINTx_EDGE_SET) registers, 9-58
pin interrupt invert clear (PINTx_INVERT_CLEAR) registers, 9-62
pin interrupt invert set (PINTx_INVERT_SET) registers, 9-61
pin interrupt latch (PINTx_LATCH) registers, 9-57
pin interrupt mask clear (PINTx_MASK_CLEAR) registers, 9-55
pin interrupt mask set (PINTx_MASK_SET) registers, 9-54, 29-95, 29-101, 29-104
pin interrupt pin state (PINTx_PINSTATE) registers, 9-60
pin interrupt request (PINTx_REQUEST) registers, 9-56
pin interrupt x (PIQx) bits, 9-54, 9-55, 9-56, 9-57, 9-58, 9-59, 9-60, 9-61, 9-62, 29-95, 29-101, 29-104
pins, 19-1
unused, 19-17
pin terminations, SPORT, 24-10
PINTx_ASSIGN (pin assignment) registers, 9-63
PINTx_EDGE_CLEAR (pin interrupt edge clear) registers, 9-59
PINTx_EDGE_SET (pin interrupt edge set) registers, 9-58
PINTx_INVERT_CLEAR (pin interrupt invert clear) registers, 9-62
PINTx_INVERT_SET (pin interrupt invert set) registers, 9-61
PINTx_LATCH (pin interrupt latch) registers, 9-57
PINTx_MASK_CLEAR (pin interrupt mask clear) registers, 9-55
PINTx_MASK_SET (pin interrupt mask set) registers, 9-54, 29-95, 29-101, 29-104
PINTx_PINSTATE (pin interrupt pin state) registers, 9-60
PINTx_REQUEST (pin interrupt request) registers, 9-56
PIO_CSTATE (PIO mode state machine current state) bits, 21-59

PIO-DMA enable (PIO_USE_DMA) bit, 21-49
PIO_DONE_INT (PIO transfer done interrupt status) bit, 21-57
PIO_DONE_MASK (PIO transfer done interrupt mask) bit, 21-55
PIO mode state machine current state (PIO_CSTATE) bits, 21-59
PIO_START (start PIO/Reg Op) bit, 21-49
PIO_TFRCNT (PIO transfer count) bits, 21-61
PIO transfer done interrupt mask (PIO_DONE_MASK) bit, 21-55
PIO transfer done interrupt status (PIO_DONE_INT) bit, 21-57
PIO transfer in progress (PIO_XFER_ON) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
PIO_USE_DMA (PIO-DMA enable) bit, 21-49
PIO_XFER_ON (PIO transfer in progress) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
pipeline
  interlocked, 3-71
pipeline, lengths of, 7-62
pipelining
  DMA requests, 7-47
PIQx (pin interrupt x) bits, 9-54, 9-55, 9-56, 9-57, 9-58, 9-59, 9-60, 9-61, 9-62, 29-95, 29-101, 29-104
PIXC_AHEND (PIXC overlay A horizontal end) register, 28-35, 28-39, A-66
PIXC_AHSTART (PIXC overlay A horizontal start) register, 28-35, 28-39, A-66
PIXC_ATRANSP (PIXC overlay A transparency) register, 28-35, 28-41, A-66
PIXC_AVEND (PIXC overlay A vertical end) register, 28-35, 28-40, A-66
PIXC_AVSTART (PIXC overlay A vertical start) register, 28-35, 28-40, A-66
PIXC_BHEND (PIXC overlay B horizontal end) register, 28-35, 28-39, A-67
PIXC_BHSTART (PIXC overlay B horizontal start) register, 28-35, 28-39, A-67
PIXC_BTRANSP (PIXC overlay B transparency) register, 28-36, 28-41, A-67
PIXC_BVCON (PIXC B/V conversion coefficients) register, 28-36, 28-44, A-67
PIXC B/V conversion coefficients (PIXC_BVCON) register, 28-36, 28-44, A-67
PIXC_BVEND (PIXC overlay B vertical end) register, 28-36, 28-40, A-67
PIXC_BVSTART (PIXC overlay B vertical start) register, 28-36, 28-40, A-67
PIXC_CCBIAS (PIXC color conversion bias) register, 28-36, 28-45, A-67
PIXC color conversion bias (PIXC_CCBIAS) register, 28-36, 28-45, A-67
PIXC control (PIXC_CTL) register, 28-35, 28-37, A-66
PIXC_CTL (PIXC control) register, 28-35, 28-37, A-66
PIXC_EN (pixel compositor enable) bit, 28-37

PIXC_GUCON (PIXC G/U conversion coefficients) register, 28-36, 28-43, A-67
PIXC G/U conversion coefficients (PIXC_GUCON) register, 28-36, 28-43, A-67
PIXC interrupt status (PIXC_INTRSTAT) register, 28-36, 28-41, A-67
PIXC_INTRSTAT (PIXC interrupt status) register, 28-36, 28-41, A-67
PIXC lines per frame (PIXC_LPF) register, 28-35, 28-38, A-66
PIXC_LPF (PIXC lines per frame) register, 28-35, 28-38, A-66
PIXC overlay A horizontal end (PIXC_AHEND) register, 28-35, 28-39, A-66
PIXC overlay A horizontal start (PIXC_AHSTART) register, 28-35, 28-39, A-66
PIXC overlay A transparency (PIXC_ATRANSP) register, 28-35, 28-41, A-66
PIXC overlay A vertical end (PIXC_AVEND) register, 28-35, 28-40, A-66
PIXC overlay A vertical start (PIXC_AVSTART) register, 28-35, 28-40, A-66
PIXC overlay B horizontal end (PIXC_BHEND) register, 28-35, 28-39, A-67
PIXC overlay B horizontal start (PIXC_BHSTART) register, 28-35, 28-39, A-67
PIXC overlay B transparency (PIXC_BTRANSP) register, 28-36, 28-41, A-67
PIXC overlay B vertical end (PIXC_BVEND) register, 28-36, 28-40, A-67
PIXC overlay B vertical start (PIXC_BVSTART) register, 28-36, 28-40, A-67
PIXC pixels per line (PIXC_PPL) register, 28-35, 28-38, A-66
PIXC_PPL (PIXC pixels per line) register, 28-35, 28-38, A-66
PIXC_RYCON (PIXC R/Y conversion coefficients) register, 28-36, 28-42, A-67
PIXC R/Y conversion coefficients (PIXC_RYCON) register, 28-36, 28-42, A-67
PIXC_TC (PIXC transparent color) register, 28-36, 28-46, A-67
PIXC transparent color (PIXC_TC) register, 28-36, 28-46, A-67
pixel compositor enable (PIXC_EN) bit, 28-37
pixels per line value (PPL) bits, 28-38
PJx pin, 22-10
PLL, 18-1
  active mode, 18-9
  applying power to the PLL, 18-13
  block diagram, 18-3
  BYPASS bit, 18-9
  bypassing onboard regulation, 18-18
  CCLK derivation, 18-3
  changing clock ratio, 18-6
  clock dividers, 18-3
  clocking to SDRAM, 18-10
  clock multiplier ratios, 18-3
  configuration, 18-3
  control bits, 18-11
  design, 18-2
  disabled, 18-13
  divide frequency, 18-3

PLL *(continued)*
DMA access, 18-8, 18-9
dynamic power management controller (DPMC), 18-7
enabled, 18-13
enabled but bypassed, 18-9
maximum performance mode, 18-8
multiplier select (MSEL) field, 18-3
operating modes, operational characteristics, 18-7
operating mode transitions, 18-13, 18-14
PDWN bit, 18-11
PLL_LOCKED bit, 18-14
PLL_OFF bit, 18-13
PLL status (table), 18-7
power domains, 18-16
powering down core, 18-20
power savings by operating mode (table), 18-8
relocking after changes, 18-14
removing power to the PLL, 18-13
RTC interrupt, 18-10, 18-15
SCLK derivation, 18-1, 18-3
sleep mode, 18-9, 18-15
STOPCK bit, 18-11
voltage control, 18-7, 18-19
PLLCLKOE (PLL clock output enable) bit, 26-142
PLL clock output enable (PLLCLKOE) bit, 26-142
PLL control register (PLL_CTL), 18-25, 18-26
PLL_CTL (PLL control register), 18-4, 18-25, 18-26
PLL divide register (PLL_DIV), 18-26
PLL_DIV (PLL divide register), 18-5, 18-25, 18-26
PLL_LOCKCNT (PLL lock count register), 18-25, 18-27
PLL lock count register (PLL_LOCKCNT), 18-27
PLL_LOCKED bit, 18-27
PLL multiplier select (M) bits, 26-142
PLL_OFF bit, 18-26
PLL stable indicator (PLL_STABLE) bit, 26-142
PLL_STABLE (PLL stable indicator) bit, 26-142
PLL Start-Up Sequence, 29-114
PLL_STAT (PLL status register), 18-25, 18-27
PLL status register (PLL_STAT), 18-27
PMAP[3:0] field, 7-10, 7-54, 7-77
POLC bits, 15-81
polling DMA registers, 7-61
POLS bits, 15-81
Port access priority, 2-12
Port access request, arbitration priority, 2-12
port connection, SPORT, 24-8
port data clear (PORTx_CLEAR) registers, 9-52
port data (PORTx) registers, 9-51
port data set (PORTx_SET) registers, 9-52
PORT_DIR bit, 13-32
port direction clear (PORTx_DIR_CLEAR) registers, 9-49
port direction set (PORTx_DIR_SET) registers, 9-49
port F
and general-purpose timers, 10-4
and SPI, 22-4
port function enable (PORTx_FER) registers, 9-46, 29-106, 29-108
port input enable (PORTx_INEN) registers, 9-50
port pins, 22-47
port pins, test access, B-2

ports, 1-14
port x bit y (Pxy) bits, 9-46, 9-49, 9-50, 9-51, 9-52, 29-106, 29-108
PORTx_CLEAR (port data clear) registers, 9-52
PORTx_DIR_CLEAR (port direction clear) registers, 9-49
PORTx_DIR_SET (port direction set) registers, 9-49
PORTx_FER (port function enable) registers, 9-46, 29-106, 29-108
PORTx_INEN (port input enable) registers, 9-50
port x multiplexer control (PORTx_MUX) registers, 9-47
PORTx_MUX (port x multiplexer control) registers, 9-47
port x mux y (MUXy) bits, 9-47
PORTx (port data) registers, 9-51
PORTx_SET (port data set) registers, 9-52
POSITION field, 29-48
Position Register Updated interrupt enable, 29-44
Position Register Updated (PRU) interrupt event, 29-31
power dissipation, 18-16
power domains, 18-16
powering down core, 18-20
power management, 1-31, 18-1
power on bits, 27-55
power save enable (PWR_SV_E) bit, 27-56
power supply management, 18-16
PPI
  GP output, 13-3, 13-5, 13-6, 13-9, 13-10
PPI_STATUS (PPI status register), 15-98, 15-99, 21-49, 21-55, 21-56
PPI status register (PPI_STATUS), 15-98, 15-99, 21-49, 21-55, 21-56
PPL (pixel per line value) bits, 28-38
P Port, 2-4
  interface, 2-8
preamble error detected (ERR_DET) bit, 15-80
preamble error not corrected (ERR_NCOR) bit, 15-80
preboot, controlled by OTP programming, 17-4
preboot routine, 17-11
precharge command, G-18
PREFETCH instruction, 3-6, 3-41
PREN bit, 14-23
prescale[6:0] field, 23-27
prescaler, RTC, 14-2
prescaler enable register (RTC_PREN), 14-23
priorities
  peripheral DMA operations, 7-56
prioritization
  DMA, 7-54 to 7-60
  interrupts, 6-11
Priority
  L2 port access, 2-12
  Sys L2 port access, 2-12
private instructions, B-4
probes, oscilloscope, 19-18
processor block diagram, 1-4
Processor Bus Hierarchy, 2-3
processor bus hierarchy, 2-3
Processor Core and L1 Memory Block Diagram, 2-7
programmable outputs, 19-17
programming model
  cache memory, 3-5
propagation segment, CAN, 31-11
PROTOCOL_R (Rx protocol type) bits, 26-131
PROTOCOL_T (Tx protocol type) bits, 26-129
PRUEN, 29-44

PS bit, 7-113
PSSE bit, 22-21, 22-45
PSSE (slave select enable) bit, 22-45
public instructions, B-4
public JTAG scan instructions, B-6
pull-up enable (PUP_SDDAT) bit, 27-72
PULSE_HI bit, 10-16, 10-18, 10-43, 10-51
PULSE_HI toggle mode, 10-18
pulse width count and capture mode, *See* WDTH_CAP mode
pulse width modulation mode, *See* PWM_OUT mode
PUP_SDDAT3 (SDH_DATA3 pull-up enable) bit, 27-72
PUP_SDDAT (pull-up enable) bit, 27-72
PWM_CLK clock, 10-22
PWM_OUT mode, 10-13 to 10-24, 10-47
  control bit and register usage, 10-51
  error prevention, 10-48
  externally clocked, 10-22
  PULSE_HI toggle mode, 10-18
  stopping the timer, 10-23
PWM_OUT PULSE_HI toggle mode bit, 10-43
PWR_ON (power on) bits, 27-55
PWR_SV_E (power save enable) bit, 27-56
Pxy (port x bit y) bits, 9-46, 9-49, 9-50, 9-51, 9-52, 29-106, 29-108

Q

query semaphore, 19-4
quick boot, 17-43

R

RBC bit, 7-46, 7-113
RBSY bit, 22-48
RBSY flag, 22-25
RBSY (receive error) bit, 22-48
RCKFE bit, 24-35, 24-57, 24-60
RCVDATA16[15:0] field, 23-51
RCVDATA8[7:0] field, 23-50
RCVFLUSH bit, 23-39, 23-40
RCVINTLEN bit, 23-39
RCVSERV bit, 23-44, 23-45
RCVSERVM bit, 23-43
RCVSTAT[1:0] field, 23-41, 23-42
RD_DLY (read strobe delay) bits, 20-19
RD_RDY (read data ready) bit, 20-21
RDTYPE[1:0] field, 24-30, 24-57, 24-59
read data ready (RD_RDY) bit, 20-21
read strobe delay (RD_DLY) bits, 20-19
read wait request enable (RWR) bit, 27-73
READY (DMA ready) bit, 8-27
READY_PAUSE (select ready to pause) bits, 21-68
real-time clock, *See* RTC
REC bit, 31-46
receive buffer[7:0] field, 25-40
receive data[15:0] field, 24-66
receive data[31:16] field, 24-66
receive data available (RX_DAT_RDY) bit, 27-63
receive data available (RX_DAT_RDY_MASK) bit, 27-68
receive data buffer[15:0] field, 22-49
received byte count in EP0 FIFO (EP0_RX_COUNT) bits, 26-128
receive error (RBSY) bit, 22-48
receive FIFO, SPORT, 24-64
receive FIFO empty (RX_DAT_ZERO) bit, 27-63
receive FIFO empty (RX_FIFO_ZERO_MASK) bit, 27-68
receive FIFO full (RX_FIFO_FULL) bit, 27-63

receive FIFO full
(RX_FIFO_FULL_MASK) bit, 27-68
receive FIFO overrun error
(RX_OVERRUN) bit, 27-64
receive FIFO overrun error
(RX_OVERRUN_MASK) bit, 27-67
receive FIFO watermark
(RX_FIFO_STAT) bit, 27-64
receive FIFO watermark
(RX_FIFO_STAT_MASK) bit, 27-67
receive message lost interrupt, CAN, 31-26
Receive Synchronous Boundary (RSB)
field, 29-26
Receiving Control Message (CMRX) bit, 29-24
reception error, SPI, 22-25
register-based DMA, 7-17
Registers
MXVR_AADDR, A-25
MXVR_ALLOC_0, A-25
MXVR_ALLOC_1, A-25
MXVR_ALLOC_10, A-26
MXVR_ALLOC_11, A-26
MXVR_ALLOC_12, A-26
MXVR_ALLOC_13, A-26
MXVR_ALLOC_14, A-26
MXVR_ALLOC_2, A-25
MXVR_ALLOC_3, A-25
MXVR_ALLOC_4, A-25
MXVR_ALLOC_5, A-26
MXVR_ALLOC_7, A-26
MXVR_ALLOC_8, A-26
MXVR_ALLOC_9, A-26
MXVR_AP_CTL, A-32
MXVR_APRB_CURR_ADDR, A-32
MXVR_APRB_START_ADDR, A-32
MXVR_APTB_CURR_ADDR, A-32
MXVR_APTB_START_ADDR, A-32
Registers *(continued)*
MXVR_CM_CTL, A-32
MXVR_CMRB_CURR_ADDR, A-32
MXVR_CMRB_START_ADDR, A-32
MXVR_CMTB_CURR_ADDR, A-33
MXVR_CMTB_START_ADDR, A-33
MXVR_CONFIG, A-24
MXVR_DELAY, A-25
MXVR_DMA0_CONFIG, A-27
MXVR_DMA0_COUNT, A-28
MXVR_DMA0_CURR_ADDR, A-28
MXVR_DMA0_CURR_COUNT, A-28
MXVR_DMA0_START_ADDR, A-27
MXVR_DMA1_CONFIG, A-28
MXVR_DMA1_COUNT, A-28
MXVR_DMA1_CURR_ADDR, A-28
MXVR_DMA1_CURR_COUNT, A-28
MXVR_DMA1_START_ADDR, A-28
MXVR_DMA2_CONFIG, A-28
MXVR_DMA2_COUNT, A-29
MXVR_DMA2_CURR_ADDR, A-29
MXVR_DMA2_CURR_COUNT, A-29
MXVR_DMA2_START_ADDR, A-29
MXVR_DMA3_CONFIG, A-29, A-30, A-31
MXVR_DMA3_COUNT, A-29, A-30, A-31
MXVR_DMA3_CURR_ADDR, A-29, A-30, A-31, A-32
MXVR_DMA3_CURR_COUNT, A-29, A-30, A-31, A-32
MXVR_DMA3_START_ADDR, A-29, A-30, A-31
MXVR_FRAME_CNT_0, A-33
MXVR_FRAME_CNT_1, A-33
MXVR_GADDR, A-25
MXVR_INT_EN_0, A-24

Registers *(continued)*
MXVR_INT_EN_1, A-24
MXVR_INT_STAT_0, A-24
MXVR_INT_STAT_1, A-24
MXVR_LADDR, A-25
MXVR_MAX_DELAY, A-25
MXVR_MAX_POSITION, A-24
MXVR_PAT_DATA_0, A-33
MXVR_PAT_DATA_1, A-33
MXVR_PAT_EN_0, A-33
MXVR_PAT_EN_1, A-33
MXVR_PLL_CTL, A-24, A-35
MXVR_POSITION, A-24
MXVR_ROUTING_0, A-34
MXVR_ROUTING_1, A-34
MXVR_ROUTING_10, A-35
MXVR_ROUTING_11, A-35
MXVR_ROUTING_12, A-35
MXVR_ROUTING_13, A-35
MXVR_ROUTING_2, A-34
MXVR_ROUTING_3, A-34
MXVR_ROUTING_4, A-34
MXVR_ROUTING_5, A-34
MXVR_ROUTING_6, A-34
MXVR_ROUTING_7, A-34
MXVR_ROUTING_8, A-34
MXVR_ROUTING_9, A-34
MXVR_RRDB_CURR_ADDR, A-33
MXVR_RRDB_START_ADDR, A-33
MXVR_STATE_0, A-24
MXVR_STATE_1, A-24
MXVR_SYNC_LCHAN_0, A-27
MXVR_SYNC_LCHAN_1, A-27
MXVR_SYNC_LCHAN_2, A-27
MXVR_SYNC_LCHAN_3, A-27
MXVR_SYNC_LCHAN_4, A-27
MXVR_SYNC_LCHAN_5, A-27
MXVR_SYNC_LCHAN_6, A-27
MXVR_SYNC_LCHAN_7, A-27
registers
See also registers by name
ATAPI, 21-46, A-39
rotary counter, 13-24, A-91
system, A-3
RegistersMXVR_ALLOC_6, A-26
REG_RXBUFFER (device receive buffer) bits, 21-54
REG_TXBUFFER (device transmit buffer) bits, 21-53
regulator controller, switching, 18-16
remote frames, CAN, 31-22
Remote GetSource Control Message Transmission, 29-155
Remote GetSource Control Message Transmit Buffer Entry Field Offsets, 29-156
Remote Get Source In Progress (RGSIP) bit, 29-24
Remote GetSource Reception, 29-162
Remote Get Source system control message, 29-24
Remote Read and Remote Write Reception, 29-159
Remote Read Buffer, 29-16
Remote Read Buffer Field Offsets, 29-160
Remote Read Control Message Transmission, 29-144
Remote Read Control Message Transmit Buffer Entry Field Offsets, 29-145
Remote Read In Progress (RRDIP) bit, 29-25
Remote Write Complete interrupt enable, 29-46
Remote Write control message, 29-16
Remote Write Control Message Complete (RWRC) interrupt event, 29-38
Remote Write Control Message Transmission, 29-146

Remote Write Control Message Transmit Buffer Entry Field Offsets, 29-147
Remote Write In Progress (RWRIP) bit, 29-25
Remote Write Receive Enable (RWRRXEN) bit, 29-16
REP bit, 7-8, 7-47, 7-113
replacement policy, 3-37
  definition, 3-80
REQPKT (request an IN transaction) bit, 26-113
REQPKT_RH (request an IN transaction) bit, 26-123
request and IN transaction (STALL_RECEIVED_RH) bit, 26-123
request and IN transaction (STALL_RECEIVED_TH) bit, 26-117
request an IN transaction (REQPKKT_RH) bit, 26-123
request an IN transaction (REQPKT) bit, 26-113
request an IN transaction (RXSTALL_TH) bit, 26-117
request data control command, DMA, 7-42
request data urgent control command, DMA, 7-42
resampling mode (UDS_MOD) bit, 28-37
reset
  effect on memory configuration, 3-29
reset, effect on SPI, 22-20
Reset Asynchronous Packet Arbitration (RESETAP) bit, 29-76
RESET_DOUBLE, 16-40, 17-104
RESET_DOUBLE bit, 16-40
reset endpoint data toggle (CLEAR_DATATOGGLE_R) bit, 26-123
reset endpoint data toggle (CLEAR_DATATOGGLE_T) bit, 26-117
RESET_OR_BABLE_BE (reset or bable IRQ enable) bit, 26-110
RESET_OR_BABLE_B (reset or bable indicator) bit, 26-109
reset or bable indicator (RESET_OR_BABLE_B) bit, 26-109
reset or bable IRQ enable (RESET_OR_BABLE_BE) bit, 26-110
$\overline{\text{RESET}}$ pin, 17-6
resets
  core and system, 8-30, 17-145, 17-146
  core double-fault, 17-6
  core-only software, 17-6
  hardware, 17-5, 17-8
  interrupts, 17-10
  software, 17-7
  system software, 17-5
  watchdog timer, 17-5, 17-7
RESET_SOFTWARE, 16-40, 17-104
RESET_SOFTWARE bit, 16-40
RESET (USB reset) bit, 26-99
reset vector, 17-1
RESET_WDOG, 16-40, 17-104
RESET_WDOG bit, 12-5, 16-40, 16-44
Resource Allocate Control Message Transmission, 29-148
Resource Allocate Control Message Transmit Buffer Entry Field Offsets, 29-149
Resource Allocate In Progress (ALIP) bit, 29-25
Resource Allocate Reception, 29-161
Resource De-Allocate Control Message Transmission, 29-152

Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets, 29-153
Resource De-Allocate In Progress (DALIP) bit, 29-24
Resource De-Allocate Reception, 29-162
resource sharing, with semaphores, 19-3
RESP_CMD (command index of last received response) bits, 27-59
restart control command, DMA, 7-40
restart or finish control command, receive, 7-44
restart or finish control command, transmit, 7-44
restrictions
  DMA control commands, 7-43
  DMA work unit, 7-32
RESUME_BE (resume signalling IRQ enable) bit, 26-110
RESUME_B (resume signalling indicator) bit, 26-109
resume mode flag (RESUME_MODE) bit, 26-99
RESUME_MODE (resume mode flag) bit, 26-99
resume signalling indicator (RESUME_B) bit, 26-109
resume signalling IRQ enable (RESUME_BE) bit, 26-110
re-synchronization, CAN, 31-12
RETI register, 17-10
RFCS (receive FIFO count status) bit, 25-37
RFHn bit, 31-78, 31-79
RFIT (receive FIFO IRQ threshold) bit, 25-32
RFRT (receive FIFO RTS threshold) bit, 25-32
RFS pins, 24-20, 24-33
RFSR bit, 24-33, 24-34, 24-57, 24-60
RFSx signal, 24-6
RGB_FMT_EN (formatting enable) bit, 15-82
RLSBIT bit, 24-57, 24-59
RMLIF bit, 31-26, 31-50
RMLIM bit, 31-26, 31-49
RMLIS bit, 31-26, 31-49
RMLn bit, 31-72
RMPn bit, 31-71
rotary counter registers, 13-24, A-91
round robin operation, MDMA, 7-57
route Rx IRQ to INTx (RX_INTx_R) bits, 26-104
route Tx IRQ to INTx (TX_INTx_R) bits, 26-104
route USB/VBUS IRQ to INTx (USB_INTx_R) bits, 26-104
routing of interrupts, 6-3, 6-4, 6-5
ROVF bit, 24-65, 24-67
row enable width (KPAD_ROWEN) bits, 30-10
rows value pressed (KPAD_ROW) bits, 30-15
RPOLC bit, 25-50, 25-51
RRFST bit, 24-58, 24-61
RSCLKx pins, 24-32
RSCLKx signal, 24-6
RSFSE bit, 24-13, 24-58, 24-61
RSPEN bit, 24-11, 24-56, 24-57, 24-59
RSTART bit, 23-32, 23-33
RTC, 1-29, 14-1 to 14-28
  alarm clock features, 14-2
  alarm feature, 14-27
  clock rate, 14-5
  clock requirements, 14-3
  code examples, 14-24
  counters, 14-2
  digital watch features, 14-1
  disabling prescaler, 14-5
  enabling prescaler, 14-4, 14-24

RTC *(continued)*
interfaces, 14-3
interrupt structure, 14-16
prescaler, 14-2
programming model, 14-6
registers, table, 14-20
state transitions, 14-17
stopwatch, 14-2, 14-25
synchronization, 14-6
test mode, 14-5
RTC alarm register (RTC_ALARM), 14-23
RTC_ALARM (RTC alarm register), 14-20, 14-23
RTC_ICTL (RTC interrupt control register), 14-20, 14-21
RTC interrupt control register (RTC_ICTL), 14-21
RTC interrupt status register (RTC_ISTAT), 14-22
RTC_ISTAT (RTC interrupt status register), 14-20, 14-22
RTC_PREN bit, 14-5
RTC_PREN (prescaler enable register), 14-4, 14-20, 14-23
RTC_STAT (RTC status register), 14-20, 14-21
RTC status register (RTC_STAT), 14-21
RTC stopwatch count register (RTC_SWCNT), 14-22
RTC_SWCNT (RTC stopwatch count register), 14-20, 14-22
RTR bit, 31-54
RUVF bit, 24-65, 24-67
RWRCEN, 29-46
RWR (Read wait request enable) bit, 27-73
RX_ACT (data receive active) bit, 27-64
RX_ACT_MASK (data receive active) bit, 27-67
RX_COUNT (USB Rx byte count) bits, 26-129
Rx data buffer status (RXS) bit, 22-48
RX_DAT_RDY_MASK (receive data available) bit, 27-68
RX_DAT_RDY (receive data available) bit, 27-63
RX_DAT_ZERO (receive FIFO empty) bit, 27-63
RXECNT[7:0] field, 31-84
RX_FIFO_FULL_MASK (receive FIFO full) bit, 27-68
RX_FIFO_FULL (receive FIFO full) bit, 27-63
RX_FIFO_STAT_MASK (receive FIFO watermark) bit, 27-67
RX_FIFO_STAT (receive FIFO watermark) bit, 27-64
RX_FIFO_ZERO_MASK (receive FIFO empty) bit, 27-68
RX hold register, 24-65
RX_INTx_R (route Rx IRQ to INTx) bits, 26-104
RXNE bit, 24-67
RX_OVERRUN_MASK (receive FIFO overrun error) bit, 27-67
RX_OVERRUN (receive FIFO overrun error) bit, 27-64
Rx packet serviced (SERVICED_RXPKTRDY) bit, 26-113
RxPktRdy autoclear enable (AUTOCLEAR_R) bit, 26-123
RXPKTRDY (data packet receive indicator) bit, 26-113
RXPKTRDY_R (data packet in FIFO indicator) bit, 26-123
RX_POLL_INTERVAL (USB Rx poll interval) bits, 26-132

Rx protocol type (PROTOCOL_R) bits, 26-131
RXREQ signal, 25-9
RXS bit, 22-31, 22-48
RXSE bit, 24-58, 24-60
RXS (Rx data buffer status) bit, 22-48
RXSTALL_TH (request an IN transaction) bit, 26-117
RX_UNDERRUN_STAT (clear receive FIFO underrun error) bit, 27-65
RY_MULT4 (multiply row by 4) bit, 28-42
RY_TRANS (transparent color - R/Y) bits, 28-46

S

SA0 bit, 31-85
SADDR[6:0] field, 23-30
SAM bit, 31-48
SAMPLE/PRELOAD instruction, B-7
sampling, CAN, 31-12
sampling edge, SPORT, 24-35
SBL2UEN, 29-44
SB (set break) bit, 25-29
SBU2LEN, 29-44
SBUEN, 29-45
scale value[7:0] field, 11-7
scaling, of core timer, 11-7
scan paths, B-5
SCD_MSK (card detect interrupt enable) bit, 27-70
SCK signal, 22-5, 22-17, 22-19, 22-22
SCL clock divider (TWI_CLKDIV) register, 23-26
SCLK, 18-4
  derivation, 18-1
  disabling, 18-20
  status by operating mode (table), 18-8
SCLOVR bit, 23-32
SCL pin, 23-5
SCLSEN bit, 23-36
SCOMP bit, 23-44, 23-47
SCOMPM bit, 23-43
scratch[7:0] field, 25-49
scratchpad memory, and booting, 17-24
scratchpad SRAM, 3-7
SCTS (sticky CTS) bit, 25-37
SD4E (SDIO 4-bit enable) bit, 27-72
SDAOVR bit, 23-32, 23-33
SDA pin, 23-5
SDASEN bit, 23-36, 23-37
SD_CARD_DET (card detect interrupt) bit, 27-70
SD_CMD_OD (SDH command open drain) bit, 27-55
SDH_ARGUMENT (SDH argument) bits, 27-57
SDH argument (SDH_ARGUMENT) bits, 27-57
SDH_ARGUMENT (SDH argument) register, 27-52, 27-57, A-37
SDH argument (SDH_ARGUMENT) register, 27-52, 27-57, A-37
SDH_CFG (SDH configuration) register, 27-54, 27-71, A-39
SDH_CLK_CTL (SDH clock control) register, 27-52, 27-55, A-37
SDH_CLK enable (CLK_E) bit, 27-56
SDH clock control (SDH_CLK_CTL) register, 27-52, 27-55, A-37
SDH clocks enable (CLKS_EN) bit, 27-72
SDH command open drain (SD_CMD_OD) bit, 27-55
SDH_COMMAND (SDH command) register, 27-52, 27-57, A-37
SDH command (SDH_COMMAND) register, 27-52, 27-57, A-37
SDH configuration (SDH_CFG) register, 27-54, 27-71, A-39

SDH_DATA3 pull-down enable (PD_SDDAT3) bit, 27-72
SDH_DATA3 pull-up enable (PUP_SDDAT3) bit, 27-72
SDH_DATA_CNT (SDH data counter) register, 27-53, 27-62, A-38
SDH data control (SDH_DATA_CTL) register, 27-53, 27-61, A-38
SDH data counter (SDH_DATA_CNT) register, 27-53, 27-62, A-38
SDH_DATA_CTL (SDH data control) register, 27-53, 27-61, A-38
SDH data FIFO bits, 27-69
SDH data FIFO (SDH_FIFOx) registers, 27-53, A-38
SDH data length (SDH_DATA_LGTH) register, 27-53, 27-61, A-38
SDH_DATA_LGTH (SDH data length register, 27-53, 27-61, A-38
SDH data (SDH_FIFOX) register, 27-69
SDH_DATA_TIMER (SDH data timer) register, 27-53, 27-60, A-38
SDH data timer (SDH_DATA_TIMER) register, 27-53, 27-60, A-38
SDH_E_MASK (SDH exception mask) register, 27-53, 27-70, A-38
SDH_E_STATUS (SDH exception status) register, 27-53, 27-69, A-38
SDH exception mask (SDH_E_MASK) register, 27-53, 27-70, A-38
SDH exception status (SDH_E_STATUS) register, 27-53, 27-69, A-38
(SDH_FIFO_CNT (SDH FIFO counter) register, 27-68
SDH_FIFO_CNT (SDH FIFO counter) register, 27-53, A-38
SDH FIFO counter (SDH_FIFO_CNT) register, 27-53, 27-68, A-38
SDH_FIFOx (SDH data FIFO) registers, 27-53, A-38
SDH_FIFOX (SDH data) register, 27-69
SDH identification (SDH_PIDX) registers, 27-73
SDH identification (SDH_PIDx) registers, 27-54, A-39
SDH interrupt mask (SDH_MASKx) registers, 27-53, 27-66, A-38
SDH_MASKx (SDH interrupt mask) registers, 27-53, 27-66, A-38
SDH_PIDx (Peripheral ID) bit, 27-73
SDH_PIDX (SDH identification) registers, 27-73
SDH_PIDx (SDH identification) registers, 27-54, A-39
SDH power control (SDH_PWR_CTL) register, 27-52, 27-55, A-37
SDH_PWR_CTL (SDH power control) register, 27-52, 27-55, A-37
SDH_RD_WAIT_EN (SDH read wait enable) register, 27-54, 27-72, A-39
SDH read wait enable (SDH_RD_WAIT_EN) register, 27-54, 27-72, A-39
SDH reset (SD_RST) bit, 27-72
SDH_RESP_CMD (SDH response command) register, 27-52, 27-58, A-37
SDH response bits, 27-59
SDH response command (SDH_RESP_CMD) register, 27-52, 27-58, A-37
SDH response (SDH_RESPONSEX) register, 27-59
SDH response (SDH_RESPONSEx) registers, 27-52, A-37
SDH_RESPONSEX (SDH response) register, 27-59
SDH_RESPONSEx (SDH response) registers, 27-52, A-37

SDH status clear (SDH_STATUS_CLR) register, 27-53, 27-65, A-38
SDH_STATUS_CLR (SDH status clear) register, 27-53, 27-65, A-38
SDH_STATUS (SDH status) register, 27-53, 27-63, A-38
SDH status (SDH_STATUS) register, 27-53, 27-63, A-38
SDIO 4-bit enable (SD4E) bit, 27-72
SDIO_INT_DET (SDIO interrupt detect) bit, 27-70
SDIO interrupt detect (SDIO_INT_DET) bit, 27-70
SDIO interrupt enable (SDIO_MSK) bit, 27-70
SDIO interrupt moving window enable (MWE) bit, 27-72
SDIO_MSK (SDIO interrupt enable) bit, 27-70
SDIR bit, 23-31
SDRAM
  banks, 3-50
  bank size, 5-2
  memory banks, 5-3
  memory space, 5-2
  sizes supported, 3-50
SD_RST (SDH reset) bit, 27-72
seconds (1 Hz) event flag bit, 14-22
seconds (1Hz) interrupt enable bit, 14-21
seconds[5:0] field, 14-21, 14-23
secure digital host
  description of operation, 27-5
  SDH clock configuration, 27-8
  SDH interface configuration, 27-9
  SDH registers, 27-52, A-37
    SDH_CLK_CTL, A-37
    SDH_PWR_CTL, A-37
select cycle time - TDVS time (TCYC_TDVS) bits, 21-67
select data valid setup time (TDVS) bits, 21-67
select DIOR/DIOW pulsewidth (TEOC_REG) bits, 21-63
select DIOR negated pulsewidth (TKR) bits, 21-65
select DIOW data hold (TH) bits, 21-66
select DIOW negated pulsewidth (TKW) bits, 21-65
SELECTED_ENDPOINT (USB endpoint index) bits, 26-103, 26-111
select end of cycle for DMA (TEOC) bits, 21-66
select envelope time (TENV) bits, 21-66
select interlock time (TMLI) bits, 21-67
select minimum delay required for output (TZAH) bits, 21-68
select ready to pause (READY_PAUSE) bits, 21-68
select setup and hold times for TACK (TACK) bits, 21-66
select time from STROBE edge to negation of DMARQ or assertion of STOP (TSS) bits, 21-67
semaphores, 19-3
  example code, 19-4
  query, 19-4
SEN bit, 23-28, 23-29
send setup token (SETUPPKT) bit, 26-113
send STALL handshake (SENDSTALL) bit, 26-113
send STALL handshake (STALL_SEND_R) bit, 26-123
send STALL handshake (STALL_SEND_T) bit, 26-117
SENDSTALL (send STALL handshake) bit, 26-113
send zero (SZ) bit, 22-45
SER bit, 31-85

serial clock divide modulus[15:0] field, 24-68
serial clock frequency, 22-22
serial communications, 25-6
serial data transfer, 24-4
serial scan paths, B-5
SERR bit, 23-44, 23-47
SERRM bit, 23-43
SERVICED_RXPKTRDY (Rx packet serviced) bit, 26-113
SERVICED_SETUPEND (setup end serviced) bit, 26-113
session end/disconnect indicator (DISCON_B) bit, 26-109
session end/disconnect IRQ enable (DISCON_BE) bit, 26-110
session indicator (SESSION) bit, 26-134, 26-136
SESSION_REQ_BE (session request IRQ enable) bit, 26-110
SESSION_REQ_B (session request indicator) bit, 26-109
session request indicator (SESSION_REQ_B) bit, 26-109
session request IRQ enable (SESSION_REQ_BE) bit, 26-110
SESSION (session indicator) bit, 26-134, 26-136
set associative (definition), 3-81
set (definition), 3-80
setup end serviced (SERVICED_SETUPEND) bit, 26-113
SETUPEND (setup end) bit, 26-113
setup end (SETUPEND) bit, 26-113
SETUPPKT (send setup token) bit, 26-113
shared interrupts, 6-11
shorten startup counter chain (TM_SHORT_CHAIN) bit, 26-142
SIC_IAR0 (system interrupt assignment register 0), 6-26, 8-25, 8-27, 8-29
SIC_IAR10 (system interrupt assignment register 10), 6-31
SIC_IAR11 (system interrupt assignment register 11), 6-31
SIC_IAR1 (system interrupt assignment register 1), 6-26
SIC_IAR2 (system interrupt assignment register 2), 6-27
SIC_IAR3 (system interrupt assignment register 3), 6-27
SIC_IAR4 (system interrupt assignment register 4), 6-28
SIC_IAR5 (system interrupt assignment register 5), 6-28
SIC_IAR6 (system interrupt assignment register 6), 6-29
SIC_IAR7 (system interrupt assignment register 7), 6-29
SIC_IAR8 (system interrupt assignment register 8), 6-30
SIC_IAR9 (system interrupt assignment register 9), 6-30
SIC_IMASK0 (system interrupt mask register), 6-32
SIC_IMASK1 (system interrupt mask register), 6-33
SIC_IMASK2 (system interrupt mask register), 6-34
SIC_IMASK (system interrupt mask register), 6-11
SIC_ISR0 (system interrupt status register), 6-35
SIC_ISR1 (system interrupt status register), 6-36
SIC_ISR2 (system interrupt status register), 6-37
SIC_ISR (system interrupt status register), 6-12

SIC_IWR0 (system interrupt wakeup register), 6-38
SIC_IWR1 (system interrupt wakeup register), 6-39
SIC_IWR2 (system interrupt wakeup register), 6-40
SIC_IWR (system interrupt wakeup-enable register), 6-13
signal integrity, 19-14
SIGN_EXT (sign extension/zero-filled) bit, 15-82
Single cast Transmission Status Encodings, 29-141
single pulse generation, timer, 10-15
single shot transmission, CAN, 31-15
SINIT bit, 23-44, 23-47
SINITM bit, 23-43
SIZE bit, 22-21, 22-45
size of accesses, timer registers, 10-37
size of words (SIZE) bit, 22-45
SIZE (size of words) bit, 22-45
SJW[1:0] field, 31-12, 31-48
SKIP_EN (skip enable) bit, 15-82
SKIP_EO (skip even/odd) bit, 15-82
Slave Mode, 29-14
Slave mode initialization, 29-112
slaves
  EBIU, 5-7
slave select, SPI, 22-46
slave select enable (FLSx) bits, 22-46
slave select enable (PSSE) bit, 22-45
slave SPI device, 22-6
sleep mode, 1-32, 18-9
  CAN, 31-39
SLEN[4:0] field, 24-52, 24-54, 24-58, 24-60
  restrictions, 24-30
  word length formula, 24-30
small descriptor mode, DMA, 7-22
small model mode, DMA, 7-82
SMODE_B (switch charge pump mode) bits, 26-142
SMR bit, 31-45
SOF_BE (start of frame IRQ enable) bit, 26-110
SOF_B (start of frame indicator) bit, 26-109
soft connect enable (SOFTC_CONN) bit, 26-99
SOFT_CONN (soft connect enable) bit, 26-99
SOFT_RST (soft reset) bit, 21-49
software interrupts, 6-10
software management of DMA, 7-60
software programmable force evaluate (KPAD_SOFTEVAL_E) bit, 30-20
software reset, 17-7, 17-103
software reset, and CAN, 31-13
software reset register (SWRST), 16-40, 17-104
software reset (SRS) bit, 31-45
software watchdog timer, 1-30, 12-1
source channels, memory DMA, 7-13
SOVF bit, 23-44, 23-46
SOVFM bit, 23-43
SPE bit, 22-21, 22-45
SPE (SPI enable) bit, 22-45
SPI, 1-20, 22-1 to 22-58
  beginning and ending transfers, 22-31
  bit mapping to port pins, 22-10
  block diagram, 22-3
  clock phase, 22-17, 22-18, 22-22
  clock polarity, 22-17, 22-22
  clock signal, 22-3, 22-22
  code examples, 22-49
  data corruption, avoiding, 22-19
  data interrupt, 22-25
  data transfer, 22-20
  detecting transfer complete, 22-23
  and DMA, 22-14

SPI *(continued)*
DMA initialization, 22-53
DMA transfers, 22-53
effect of reset, 22-20
error interrupt, 22-25
error signals, 22-23 to 22-25
general operation, 22-26 to 22-30
initialization, 22-50
internal interfaces, 22-14
interrupt outputs, 22-25
interrupts, 22-52
master mode, 22-20, 22-26
master mode DMA operation, 22-33
mode fault error, 22-24
multiple slave systems, 22-12
port F, 22-4
reception error, 22-25
registers, table, 22-43
SCK signal, 22-5
slave boot mode, 17-73
slave device, 22-6
slave mode, 22-20, 22-29
slave mode DMA operation, 22-35
slave select enable setup, 22-2, 22-9
slave-select function, 22-46
slave transfer preparation, 22-30
SPI_FLG mapping to port pins, 22-47
starting DMA transfer, 22-56
starting transfer, 22-51
stopping, 22-53
stopping DMA transfers, 22-56
switching between transmit and receive, 22-32
timing, 22-8
transfer formats, 22-17 to 22-18
transfer initiate command, 22-27
transfer modes, 22-28
transfer protocol, 22-18, 22-19
transmission error, 22-25
transmission/reception errors, 22-23
SPI *(continued)*
transmit collision error, 22-25
using DMA, 22-14
word length, 22-21
SPI baud rate registers (SPI_BAUD), 22-22, 22-43
SPI baud rate (SPIx_BAUD) registers, 22-44
SPI_BAUD (SPI baud rate registers), 22-22, 22-43
SPI_BAUD values, 22-23
SPI control register (SPI_CTL), 22-21, 22-43, 22-45
SPI control (SPIx_CTL) registers, 22-45
SPI_CTL (SPI control register), 22-5, 22-21, 22-43, 22-45
SPI enable (SPE) bit, 22-45
SPIF bit, 22-13, 22-31, 22-48
SPI finished (SPIF) bit, 22-48
SPI flag register (SPI_FLG), 22-10, 22-43, 22-46
SPI flag (SPIx_FLG) registers, 22-46
SPI_FLG bit, 22-10
SPI_FLG (SPI flag register), 22-10, 22-12, 22-43, 22-46
SPIF (SPI finished) bit, 22-48
SPI_RDBR shadow[15:0] field, 22-49
SPI RDBR shadow register (SPI_SHADOW), 22-16, 22-43, 22-49
SPI RDBR shadow (SPIx_SHADOW) registers, 22-49
SPI_RDBR (SPI receive data buffer register), 22-16, 22-43, 22-49
SPI receive data buffer register (SPI_RDBR), 22-16, 22-43, 22-49
SPI receive data buffer (SPIx_RDBR) registers, 22-49
SPI_SHADOW (SPI RDBR shadow register), 22-16, 22-43, 22-49

SPI slave select, 22-46
SPISS signal, 22-7, 22-12, 22-17
SPI_STAT (SPI status register), 22-23, 22-43, 22-48
SPI status register (SPI_STAT), 22-23, 22-43, 22-48
SPI status (SPIx_STAT) registers, 22-48
SPI_TDBR (SPI transmit data buffer register), 22-15, 22-43, 22-48
SPI transmit data buffer register (SPI_TDBR), 22-15, 22-43, 22-48
SPI transmit data buffer (SPIx_TDBR) registers, 22-48
SPIx_BAUD (SPI baud rate) registers, 22-44
SPIx_CTL (SPI control) registers, 22-45
SPIx_FLG (SPI flag) registers, 22-46
SPIx_RDBR (SPI receive data buffer) registers, 22-49
SPIx_SHADOW (SPI RDBR shadow) registers, 22-49
SPIx_STAT (SPI status) registers, 22-48
SPIx_TDBR data buffer status (TXS) bit, 22-48
SPIx_TDBR (SPI transmit data buffer) registers, 22-48
SPLT_EVEN_ODD, 15-82
SPORT, 1-19, 24-1 to 24-82
  active low vs. active high frame syncs, 24-35
  channels, 24-17
  clock, 24-32
  clock frequency, 24-28, 24-68
  clock rate, 24-2
  clock rate restrictions, 24-29
  clock recovery control, 24-27
  companding, 24-31
  configuration, 24-12
  data formats, 24-30
  data word formats, 24-61
SPORT *(continued)*
  disabling, 24-12, 24-32
  DMA data packing, 24-26
  enable/disable, 24-11
  enabling multichannel mode, 24-19
  framed serial transfers, 24-34
  framed vs. unframed, 24-33
  frame sync, 24-34, 24-38
  frame sync frequencies, 24-28
  framing signals, 24-33
  general operation, 24-11
  H.100 standard protocol, 24-27
  initialization code, 24-59
  internal memory access, 24-40
  internal vs. external frame syncs, 24-34
  late frame sync, 24-19
  modes, 24-12
  moving data to memory, 24-40
  multichannel frame, 24-22
  multichannel operation, 24-17 to 24-27
  multichannel transfer timing, 24-18
  multiplexed pins, 24-4
  PAB error, 24-41
  packing data, multichannel DMA, 24-26
  pins, 24-4
  port connection, 24-8
  port G, 24-4
  receive and transmit functions, 24-4
  receive clock signal, 24-32
  receive FIFO, 24-64
  receive word length, 24-65
  register writes, 24-50
  RX hold register, 24-65
  sampling edge, 24-35
  selecting bit order, 24-30
  serial data communication protocols, 24-1
  shortened active pulses, 24-12, 24-32
  signals, 24-5

SPORT *(continued)*
single clock for both receive and transmit, 24-32
single word transfers, 24-40
stereo serial connection, 24-10
stereo serial frame sync modes, 24-19
stereo serial operation, 24-13
support for standard protocols, 24-27
termination, 24-10
timing, 24-42
transmit clock signal, 24-32
transmitter FIFO, 24-61
transmit word length, 24-62
TX hold register, 24-62
TX interrupt, 24-62
unframed data flow, 24-33
unpacking data, multichannel DMA, 24-26
window offset, 24-24
word length, 24-30
SPORT current channel (SPORTx_CHNL) registers, 24-49, 24-71
SPORT error interrupt, 24-41
SPORT multichannel configuration (SPORTx_MCMC1) register 1, 24-49
SPORT multichannel configuration (SPORTx_MCMC2) register 2, 24-49
SPORT multichannel receive select (SPORTx_MRCSn) registers, 24-49, 24-72
SPORT multichannel transmit select registers (SPORTx_MTCSn), 24-25
SPORT multichannel transmit select (SPORTx_MTCSn) registers, 24-49, 24-74
SPORT receive configuration 1 (SPORTx_RCR1) registers, 24-48, 24-56
SPORT receive configuration 2 (SPORTx_RCR2) registers, 24-48, 24-56
SPORT receive data (SPORTx_RX) registers, 24-48, 24-64, 24-65
SPORT receive frame sync divider (SPORTx_RFSDIV) registers, 24-48, 24-69
SPORT receive serial clock divider (SPORTx_RCLKDIV) registers, 24-48, 24-68
SPORT RX interrupt, 24-41, 24-65
SPORT status (SPORTx_STAT) registers, 24-48, 24-66
SPORT transmit configuration 1 (SPORTx_TCR1) registers, 24-48, 24-51
SPORT transmit configuration 2 (SPORTx_TCR2) registers, 24-48, 24-51
SPORT transmit data (SPORTx_TX) registers, 24-48, 24-61, 24-63
SPORT transmit frame sync divider (SPORTx_TFSDIV) registers, 24-48, 24-69
SPORT transmit serial clock divider (SPORTx_TCLKDIV) registers, 24-48, 24-68
SPORT TX interrupt, 24-41
SPORTx_CHNL (SPORT current channel) registers, 24-49, 24-71
SPORTx_MCMC1 (SPORT multichannel configuration) register 1, 24-49
SPORTx_MCMC2 (SPORT multichannel configuration) register 2, 24-49

SPORTx_MRCSn (SPORT multichannel receive select) registers, 24-49, 24-72
SPORTx_MTCSn (SPORT multichannel transmit select) registers, 24-49, 24-74
SPORTx multichannel configuration registers (SPORTx_MCMCn), 24-70
SPORTx multichannel receive select registers (SPORTx_MRCSn), 24-25, 24-26
SPORTx multichannel transmit select registers (SPORTx_MTCSn), 24-25
SPORTx_RCLKDIV (SPORT receive serial clock divider) registers, 24-48, 24-68
SPORTx_RCR1 (SPORT receive configuration 1) registers, 24-48, 24-56
SPORTx_RCR2 (SPORT receive configuration 2) registers, 24-48, 24-56
SPORTx_RCR2 (SPORTx receive configuration register), 24-58
SPORTx receive configuration 2 registers (SPORTx_RCR2), 24-58
SPORTx receive data registers (SPORTx_RX), 24-21
SPORTx_RFSDIV (SPORT receive frame sync divider) registers, 24-48, 24-69
SPORTx_RX (SPORT receive data) registers, 24-48, 24-64, 24-65
SPORTx_STAT (SPORT status) registers, 24-48, 24-66
SPORTx_TCLKDIV (SPORT transmit serial clock divider) registers, 24-48, 24-68
SPORTx_TCR1 (SPORT transmit configuration 1) registers, 24-48, 24-51
SPORTx_TCR2 (SPORT transmit configuration 2) registers, 24-48, 24-51
SPORTx_TFSDIV (SPORT transmit frame sync divider) registers, 24-48, 24-69
SPORTx transmit data registers (SPORTx_TX), 24-21, 24-40
SPORTx_TX (SPORT transmit data) registers, 24-48, 24-61, 24-63
SRAM
  interface, 19-8
  L1 data, 3-30
  L1 Data Memory, 3-7
  L1 instruction access, 3-11
  L1 Instruction Memory, 3-6
  scratchpad, 3-7
SRS bit, 31-45
SRS (software reset) bit, 31-45
SSEL[3:0] field, 18-5, 18-26
STALL handshake received (STALL_RECEIVED) bit, 26-113
STALL handshake sent (STALL_SENT) bit, 26-113
STALL handshake sent (STALL_SENT_R) bit, 26-123
STALL handshake sent (STALL_SENT_T) bit, 26-117
STALL_RECEIVED_RH (request and IN transaction) bit, 26-123
STALL_RECEIVED (STALL handshake received) bit, 26-113
STALL_RECEIVED_TH (request and IN transaction) bit, 26-117
Stalls
  DMA access to L1 or L2 memory, 2-13
stalls
  pipeline, 3-71
STALL_SEND_R (send STALL handshake) bit, 26-123

STALL_SEND_T (send STALL handshake) bit, 26-117
STALL_SENT_R (STALL handshake sent) bit, 26-123
STALL_SENT (STALL handshake sent) bit, 26-113
STALL_SENT_T (STALL handshake sent) bit, 26-117
start address registers
(DMAx_START_ADDR), 7-88
(MDMA_yy_START_ADDR), 7-88
Start Asynchronous Packet Transmission (STARTAP) bit, 29-75
START_BIT_ERR_MASK (start bit error) bit, 27-67
start bit error (START_BIT_ERR) bit, 27-64
start bit error (START_BIT_ERR_MASK) bit, 27-67
START_BIT_ERR (start bit error) bit, 27-64
START_BIT_ERR_STAT (clear start bit error) bit, 27-65
Start Control Message Transmission (STARTCM) bit, 29-80
start of frame indicator (SOF_B) bit, 26-109
start of frame IRQ enable (SOF_B) bit, 26-110
Start Pattern select (STARTPATx) field, 29-67
state transitions, RTC, 14-17
Status Change Interrupt, 29-30, 29-31
status (CNT_STATUS) register, 13-28
STATUSPKT_H (packet transaction status) bit, 26-113
STB (stop bits) bit, 25-29
STDVAL bit, 23-28, 23-29
stereo serial data, 24-2
stereo serial device, SPORT connection, 24-10
stereo serial frame sync modes, 24-19
stereo serial operation, SPORT, 24-13
STI. *See* Enable Interrupts (STI)
STOP bit, 23-34
STOPCK bit, 18-26
Stop Mode, 29-62, 29-72
stop mode, DMA, 7-18, 7-82
Stop Pattern select (STOPPATx) field, 29-67
stopping DMA transfers, 7-36
stopwatch count[15:0] field, 14-22
stopwatch function, RTC, 14-2
store operation, 3-70
store ordering, 3-72
STP (stick parity) bit, 25-29
strong ordering requirement, 3-78
subbanks
L1 instruction memory, 3-11
SUBSPLT_ODD (sub-split odd samples) bit, 15-82
Super Block Locked State (SBLOCK) bit, 29-31
Super Block Locked to Unlocked interrupt enable, 29-44
Super Block Locked to Unlocked (SBL2U) interrupt event, 29-31
Super Block Lock (SBLOCK) bit, 29-20
Super Block Unlocked to Locked interrupt enable, 29-44
Super Block Unlocked to Locked (SBU2L) interrupt event, 29-31
supervisor mode, 17-10
SUSPEND_BE (suspend signalling IRQ enable) bit, 26-110
SUSPEND_B (suspend signalling indicator) bit, 26-109
suspend mode, CAN, 31-39

suspend mode enable (SUSPEND_MODE) bit, 26-99
suspend mode output enable (ENABLE_SUSPENDM) bit, 26-99
SUSPEND_MODE (suspend mode enable) bit, 26-99
suspend signalling indicator (SUSPEND_B) bit, 26-109
suspend signalling IRQ enable (SUSPEND_BE) bit, 26-110
SWAPEN (swap enable) bit, 15-82
switch charge pump mode (SMODE_B) bits, 26-142
switching frequency values, 18-18
switching regulator controller, 18-16
SWRESET, 16-58, 16-59, 17-106
SWRST, software reset register, 17-103
SWRST (software reset register), 16-40, 17-104
SYNC, 3-74
SYNC bit, 7-33, 7-35, 7-72, 7-80, 7-83, 25-25
synchronization
  interrupt-based methods, 7-61
  of descriptor queue, 7-67
  of DMA, 7-60 to 7-70
synchronized transition, DMA, 7-35
Synchronous Boundary (MSB) field, 29-18
Synchronous Boundary Updated interrupt enable, 29-45
Synchronous Boundary Updated (SBU) interrupt event, 29-33
Synchronous Data Delay (SDELAY) bit, 29-14
Synchronous Data Interrupt, 29-41
Synchronous Data Reception, 29-125
Synchronous Data Routing, Muting, and Transmission, 29-121
Synchronous Packet Autobuffer Modes, 29-63
Synchronous Packet-Fixed Count Mode, 29-70, 29-73, 29-74
Synchronous Packet-Start/Stop Mode, 29-64, 29-70, 29-73, 29-75
Synchronous Packet-Variable Count Mode, 29-63, 29-64, 29-70, 29-73, 29-74
Synchronous Receive FIFO Number of Bytes (SRXNUMB) field, 29-27
Synchronous Receive FIFO Number of Bytes (STXNUMB) field, 29-27
synchronous serial data transfer, 24-4
synchronous serial ports, *See* SPORT
SYSCR (System Reset Configuration Register), 17-106
SYSCR (system reset configuration register), 16-58, 16-59, 16-65, 16-67, 17-105
System
  DMA access request, 2-11
  L2 bus, 2-5
  overview, 2-8
system clock (SCLK), 18-1
  managing, 19-2
system design, 19-1 to 19-19
  high frequency considerations, 19-5
  recommendations and suggestions, 19-15
  recommended reading, 19-19
system interrupt assignment register 0 (SIC_IAR0), 6-26, 8-25, 8-27, 8-29
system interrupt assignment register 10 (SIC_IAR10), 6-31
system interrupt assignment register 1 (SIC_IAR1), 6-26, 6-31
system interrupt assignment register 2 (SIC_IAR2), 6-27
system interrupt assignment register 3 (SIC_IAR3), 6-27

system interrupt assignment register 4 (SIC_IAR4), 6-28
system interrupt assignment register 5 (SIC_IAR5), 6-28
system interrupt assignment register 6 (SIC_IAR6), 6-29
system interrupt assignment register 7 (SIC_IAR7), 6-29
system interrupt assignment register 8 (SIC_IAR8), 6-30
system interrupt assignment register 9 (SIC_IAR9), 6-30
system interrupt controller (SIC), 6-2, 6-6
  registers, 6-24
system interrupt mask register (SIC_IMASK), 6-11
system interrupt mask register (SIC_IMASK0), 6-32
system interrupt mask register (SIC_IMASK1), 6-33
system interrupt mask register (SIC_IMASK2), 6-34
system interrupt processing, 6-22
system interrupts, 6-6
system interrupt status register (SIC_ISR), 6-12
system interrupt status register (SIC_ISR0), 6-35
system interrupt status register (SIC_ISR1), 6-36
system interrupt status register (SIC_ISR2), 6-37
system interrupt wakeup-enable register (SIC_IWR), 6-13
system interrupt wakeup register (SIC_IWR0), 6-38
system interrupt wakeup register (SIC_IWR1), 6-39
system interrupt wakeup register (SIC_IWR2), 6-40
system peripheral clock, *See* SCLK
system peripherals, 1-2
SYSTEM_RESET, 16-40, 17-104
SYSTEM_RESET[2:0] field, 16-40, 16-44
System Reset Configuration Register (SYSCR), 17-106
system reset configuration register (SYSCR), 16-58, 16-59, 16-65, 16-67, 17-105
system software reset, 17-5
SZ bit, 22-30, 22-45
SZ (send zero) bit, 22-45

T

T1_REG (time from address valid to DIOR/DIOW) bits, 21-64
T2_REG (end of cycle time for register access transfers) bits, 21-63
T2_REG_PIO (DIOR/DIOW pulsewidth) bits, 21-64
T4_REG (DIOW data hold) bits, 21-64
TACIx pins, 10-5, 10-33
TACK (select setup and hold times for TACK) bits, 21-66
TACLKx pins, 10-5
tag (definition), 3-81
TAn bit, 31-77
TAP registers
  boundary-scan, B-8
  bypass, B-7
  instruction, B-2, B-4
TAP (test access port), B-2
  controller, B-2
target address, 17-31
TARGET_EP_NO_R (target EPx number) bits, 26-131
TARGET_EP_NO_T (target EPx number) bits, 26-129
target EPx number (TARGET_EP_NO_R) bits, 26-131

target EPx number
(TARGET_EP_NO_T) bits, 26-129
TAUTORLD bit, 11-3, 11-5
TC_EN (transparent color enable) bit, 28-37
TCKFE bit, 24-35, 24-51, 24-55
TCNTL (core timer control register), 11-3, 11-5
TCOUNT (core timer count register), 11-3, 11-5
TCYC_TDVS (select cycle time - TDVS time) bits, 21-67
TDA bit, 31-23, 31-78
TD (DIOR/DIOW asserted pulsewidth) bits, 21-65
TDM interfaces, 24-3
TDPTR[4:0] field, 31-78
TDR bit, 31-23, 31-78
TDTYPE[1:0] field, 24-30, 24-51, 24-53
TDVS (select data valid setup time) bits, 21-67
technical support, lxxxiii
TEMT bit, 25-8, 25-35, 25-36
TENV (select envelope time) bits, 21-66
TEOC_REG_PIO (end of cycle time for PIO access transfers) bits, 21-64
TEOC_REG (select DIOR/DIOW pulsewidth) bits, 21-63
TEOC (select end of cycle for DMA) bits, 21-66
termination, DMA, 7-36
terminations, SPORT pin/line, 24-10
test access port (TAP), B-2
controller, B-2
test clock (TCK), B-7
test features, B-1 to B-8
testing circuit boards, B-1, B-6
test-logic-reset state, B-3
test point access, 19-18
TESTSET instruction, 19-3
TFI (transmission finished indicator) bit, 25-35, 25-37
TFRCNT_RST (transmission count reset) bit, 21-49
TFS pins, 24-33, 24-40
TFSR bit, 24-33, 24-34, 24-51, 24-54
TFS signal, 24-21
TFSx signal, 24-6
THRE bit, 25-16, 25-36
THRE flag, 25-7, 25-23
THRE (transmit hold register empty) bit, 25-35
throughput
achieved by interlocked pipeline, 3-71
achieved by SRAM, 3-5
DMA, 7-51
from DMA system, 7-50
SPORT, 24-8
Throughput for L2 memory access, 2-13
TH (select DIOW data hold) bits, 21-66
TIMDIS0 bit, 10-39
TIMDIS10 bit, 10-39
TIMDIS1 bit, 10-39
TIMDIS2 bit, 10-39
TIMDIS3 bit, 10-39
TIMDIS4 bit, 10-39
TIMDIS5 bit, 10-39
TIMDIS6 bit, 10-39
TIMDIS7 bit, 10-39
TIMDIS8 bit, 10-39
TIMDIS9 bit, 10-39
TIMDISx bit, 10-39
time-division-multiplexed (TDM) mode, 24-17
See also SPORT, multichannel operation
time from address valid to DIOR/DIOW (T1_REG) bits, 21-64
time from address valid to DIOR/DIOW (TM) bits, 21-65
TIMEN0 bit, 10-38

TIMEN10 bit, 10-38
TIMEN1 bit, 10-38
TIMEN2 bit, 10-38
TIMEN3 bit, 10-38
TIMEN4 bit, 10-38
TIMEN5 bit, 10-38
TIMEN6 bit, 10-38
TIMEN7 bit, 10-38
TIMEN8 bit, 10-38
TIMEN9 bit, 10-38
TIMENx bit, 10-38
timeout error (ERROR_H) bit, 26-113
timeout error indicator (ERROR_RH) bit, 26-123
timeout error indicator (ERROR_TH) bit, 26-117
TIMEOUT (host timeout) bit, 8-27
Timer 0 counter overflow bit, 10-41
Timer 0 disable bit, 10-39
Timer 0 enable bit, 10-38
Timer 0 interrupt bit, 10-41
Timer 0 slave enable status bit, 10-41
Timer 10 counter overflow bit, 10-42
Timer 10 disable bit, 10-39
Timer 10 enable bit, 10-38
Timer 10 interrupt bit, 10-42
Timer 10 slave enable status bit, 10-42
Timer 1 counter overflow bit, 10-41
Timer 1 disable bit, 10-39
Timer 1 enable bit, 10-38
Timer 1 interrupt bit, 10-41
Timer 1 slave enable status bit, 10-41
Timer 2 counter overflow bit, 10-41
Timer 2 disable bit, 10-39
Timer 2 enable bit, 10-38
Timer 2 interrupt bit, 10-41
Timer 2 slave enable status bit, 10-41
Timer 3 counter overflow bit, 10-41
Timer 3 disable bit, 10-39
Timer 3 enable bit, 10-38
Timer 3 interrupt bit, 10-41
Timer 3 slave enable status bit, 10-41
Timer 4 counter overflow bit, 10-41
Timer 4 disable bit, 10-39
Timer 4 enable bit, 10-38
Timer 4 interrupt bit, 10-41
Timer 4 slave enable status bit, 10-41
Timer 5 counter overflow bit, 10-41
Timer 5 disable bit, 10-39
Timer 5 enable bit, 10-38
Timer 5 interrupt bit, 10-41
Timer 5 slave enable status bit, 10-41
Timer 6 counter overflow bit, 10-41
Timer 6 disable bit, 10-39
Timer 6 enable bit, 10-38
Timer 6 interrupt bit, 10-41
Timer 6 slave enable status bit, 10-41
Timer 7 counter overflow bit, 10-41
Timer 7 disable bit, 10-39
Timer 7 enable bit, 10-38
Timer 7 interrupt bit, 10-41
Timer 7 slave enable status bit, 10-41
Timer 8 counter overflow bit, 10-42
Timer 8 disable bit, 10-39
Timer 8 enable bit, 10-38
Timer 8 interrupt bit, 10-42
Timer 8 slave enable status bit, 10-42
Timer 9 counter overflow bit, 10-42
Timer 9 disable bit, 10-39
Timer 9 enable bit, 10-38
Timer 9 interrupt bit, 10-42
Timer 9 slave enable status bit, 10-42
timer clock select bit, 10-43
timer configuration registers (TIMERx_CONFIG), 10-42, 10-43
timer counter[15:0] field, 10-45
timer counter[31:16] field, 10-45
timer counter registers (TIMERx_COUNTER), 10-44, 10-45

TIMER_DISABLE0 (timer disable register), 10-39
TIMER_DISABLE1 (timer disable register), 10-39
TIMER_DISABLE bit, 10-51
timer disable register (TIMER_DISABLE), 10-39
timer disable register (TIMER_DISABLE0), 10-39
timer disable register (TIMER_DISABLE1), 10-39
TIMER_DISABLE (timer disable register), 10-39
TIMER_ENABLE0 (timer enable register), 10-38
TIMER_ENABLE1 (timer enable register), 10-38
TIMER_ENABLE bit, 10-51
timer enable register (TIMER_ENABLE), 10-38
timer enable register (TIMER_ENABLE0), 10-38
timer enable register (TIMER_ENABLE1), 10-38
TIMER_ENABLE (timer enable register), 10-38
timer input select bit, 10-43
Timer mode field, 10-43
timer period[15:0] field, 10-49
timer period[31:16] field, 10-49
timer period registers (TIMERx_PERIOD), 10-47, 10-49
timers, 1-21, 10-1 to 10-62
  core, 11-1 to 11-8
  EXT_CLK mode, 10-34 to 10-35
  watchdog, 1-30, 12-1 to 12-11
  WDTH_CAP mode, 25-22
TIMER_STATUS0 (timer status register), 10-41
TIMER_STATUS1 (timer status register), 10-42
timer status register (TIMER_STATUS), 10-40
timer status register (TIMER_STATUS0), 10-41
timer status register (TIMER_STATUS1), 10-42
TIMER_STATUS (timer status register), 10-40
timer width[15:0] field, 10-50
timer width[31:16] field, 10-50
timer width registers (TIMERx_WIDTH), 10-47, 10-50
TIMERx_CONFIG (timer configuration registers), 10-42, 10-43
TIMERx_COUNTER (timer counter registers), 10-6, 10-44, 10-45
TIMERx_PERIOD (timer period registers), 10-47, 10-49
TIMERx_WIDTH (timer width registers), 10-47, 10-50
time stamps, CAN, 31-21
TIMIL0 bit, 10-41
TIMIL10 bit, 10-42
TIMIL1 bit, 10-41
TIMIL2 bit, 10-41
TIMIL3 bit, 10-41
TIMIL4 bit, 10-41
TIMIL5 bit, 10-41
TIMIL6 bit, 10-41
TIMIL7 bit, 10-41
TIMIL8 bit, 10-42
TIMIL9 bit, 10-42
TIMILx bits, 10-6
timing
  memory DMA, 7-54
  multichannel transfer, 24-18
  SPI, 22-8
timing examples, for SPORTs, 24-42

timing parameters, CAN, 31-12
TIMOD[1:0] field, 22-21, 22-26, 22-28, 22-45
TIMODx (transfer initiation mode) bits, 22-45
TIN_SEL bit, 10-33, 10-43, 10-51
TINT bit, 11-3, 11-5
TKR (select DIOR negated pulsewidth) bits, 21-65
TKW (select DIOW negated pulsewidth) bits, 21-65
TLSBIT bit, 24-51, 24-54
TMLI (select interlock time) bits, 21-67
TMODE[1:0] field, 10-13, 10-43, 10-51
TM_PLL_VCO (boost PLL amplitude) bit, 26-142
TMPWR bit, 11-3, 11-5
TMRCLK input, 10-5
TMREN bit, 11-3, 11-5
TMR pin, 10-52
TMRx pins, 10-4, 10-17, 10-33
TM_SELC (increase PLL charge pump current) bit, 26-142
TM_SHORT_CHAIN (shorten startup counter chain) bit, 26-142
TM (time from address valid to DIOR/DIOW) bits, 21-65
TOGGLE_HI bit, 10-43, 10-52
TOGGLE_HI mode, 10-18
tools, development, 1-36
TOVF bit, 24-63, 24-67
TOVF_ERR0 bit, 10-41
TOVF_ERR10 bit, 10-42
TOVF_ERR1 bit, 10-41
TOVF_ERR2 bit, 10-41
TOVF_ERR3 bit, 10-41
TOVF_ERR4 bit, 10-41
TOVF_ERR5 bit, 10-41
TOVF_ERR6 bit, 10-41
TOVF_ERR7 bit, 10-41
TOVF_ERR8 bit, 10-42
TOVF_ERR9 bit, 10-42
TOVF_ERRx bits, 10-6, 10-10, 10-17, 10-42, 10-53
TPERIOD (core timer period register), 11-6
TPOLC bit, 25-50, 25-51
traffic control, DMA, 7-54 to 7-60
Traffic control/optimization, DMA, 2-13
transfer count (ECCCNT) bits, 20-24
transfer direction (XFER_DIR) bit, 21-49
transfer initiate command, 22-27
transfer initiation from SPI master, 22-28
transfer initiation mode (TIMODx) bits, 22-45
Transfer latency
  DMA data, 2-13
transfer length (XFER_LENGTH) bits, 21-58
transfer rate
  memory DMA channels, 7-51
  peripheral DMA channels, 7-51
transfer size (TxferSize), 26-31, 26-34
transitions
  continuous DMA, 7-32
  DMA work unit, 7-32
  operating mode, 18-11, 18-12
  synchronized DMA, 7-32
transmission count reset (TFRCNT_RST) bit, 21-49
transmission error, SPI, 22-25
transmission error (TXE) bit, 22-48
transmit clock, serial (TSCLKx) pins, 24-32
transmit collision error, SPI, 22-25
transmit collision error (TXCOL) bit, 22-48
transmit data[15:0] field, 24-63
transmit data[31:16] field, 24-63

transmit data available (TX_DAT_RDY) bit, 27-63
transmit data available (TX_DAT_RDY_MASK) bit, 27-68
transmit data buffer[15:0] field, 22-48
transmit FIFO empty (TX_FIFO_ZERO) bit, 27-63
transmit FIFO empty (TX_FIFO_ZERO_MASK) bit, 27-68
transmit FIFO full (TX_FIFO_FULL) bit, 27-63
transmit FIFO full (TX_FIFO_FULL_MASK) bit, 27-68
transmit FIFO underrun error (TX_UNDERRUN_MASK) bit, 27-67
transmit FIFO watermark (TX_FIFO_STAT) bit, 27-64
transmit FIFO watermark (TX_FIFO_STAT_MASK) bit, 27-67
Transmit FOT, 29-16
transmit hold[7:0] field, 25-39, 25-40
transmit underrun (TX_UNDERRUN) bit, 27-64
transparent color - B/V (BV_TRANS) bits, 28-46
transparent color enable (TC_EN) bit, 28-37
transparent color - G/U (GU_TRANS) bits, 28-46
transparent color - R/Y (RY_TRANS) bits, 28-46
TRFST bit, 24-52, 24-56
triggering DMA transfers, 7-71
TRM bit, 31-46
TRRn bit, 31-75
TRSn bit, 31-74
TRUN0 bits, 10-41
TRUN10 bits, 10-42
TRUN1 bits, 10-41
TRUN2 bits, 10-41
TRUN3 bits, 10-41
TRUN4 bits, 10-41
TRUN5 bits, 10-41
TRUN6 bits, 10-41
TRUN7 bits, 10-41
TRUN8 bits, 10-42
TRUN9 bits, 10-42
TRUNx bits, 10-23, 10-40, 10-52
TSCALE (core timer scale register), 11-7
TSCLKx signal, 24-6
TSEG1[3:0] field, 31-11, 31-48
TSEG2[2:0] field, 31-11, 31-48
TSFSE bit, 24-13, 24-52, 24-56
TSPEN bit, 24-11, 24-51, 24-52, 24-53
TSS (select time from STROBE edge to negation of DMARQ or assertion of STOP) bits, 21-67
TSV[15:0] field, 31-58
tuning of DPHY clocks (CNOS) bits, 26-141
TUVF bit, 24-40, 24-63, 24-66, 24-67
TWI, 1-15, 23-1 to 23-63
  block diagram, 23-3
  bus arbitration, 23-8
  clock generation, 23-7
  electrical specifications, 23-63
  fast mode, 23-10
  features, 23-2
  general call address, 23-10
  general setup, 23-13
  I^2C compatibility, 1-15
  master boot mode, 17-77
  master mode clock setup, 23-14
  master mode receive, 23-16
  master mode transmit, 23-15
  peripheral interface, 23-5

TWI *(continued)*
pins, 23-5
registers, list of, 23-25, A-13, A-14
slave boot mode, 17-79
slave mode operation, 23-13
start and stop conditions, 23-9
synchronization, 23-7
transfer protocol, 23-6
TWI_CLKDIV (SCL clock divider) register, 23-26
TWI_CONTROL (TWI control) register, 23-27
TWI_FIFO_CTL (TWI FIFO control) register, 23-39
TWI FIFO receive data double byte (TWI_RCV_DATA16 register, 23-51
TWI FIFO receive data single byte (TWI_RCV_DATA8 register, 23-50
TWI_FIFO_STAT (TWI FIFO status register), 23-41
TWI_FIFO_STAT (TWI FIFO status) register, 23-41
TWI FIFO status register (TWI_FIFO_STAT), 23-41
TWI FIFO transmit data double byte (TWI_XMT_DATA16) register, 23-49
TWI FIFO transmit data single byte (TWI_XMT_DATA8 register, 23-48
TWI interrupt mask (TWI_INT_MASK) register, 23-43
TWI interrupt status register (TWI_INT_STAT), 23-44
TWI_INT_MASK (TWI interrupt mask) register, 23-43
TWI_INT_STAT (TWI interrupt status register), 23-44
TWI_INT_STAT (TWI interrupt status) register, 23-44
TWI_MASTER_ADDR (TWI master mode address) register, 23-35
TWI master mode status register (TWI_MASTER_STAT), 23-35
TWI master mode status (TWI_MASTER_STAT) register, 23-35
TWI_MASTER_STAT (TWI master mode status register), 23-35
TWI_MASTER_STAT (TWI master mode status) register, 23-35
TWI_RCV_DATA16 (TWI FIFO receive data double byte) register, 23-51
TWI_RCV_DATA8 (TWI FIFO receive data single byte) register, 23-50
TWI_SLAVE_ADDR (TWI slave mode address) register, 23-30
TWI_SLAVE_CTL (TWI slave mode control) register, 23-27
TWI slave mode control (TWI_SLAVE_CTL) register, 23-27
TWI slave mode status (TWI_SLAVE_STAT) register, 23-30
TWI_SLAVE_STAT (TWI slave mode status) register, 23-30
TWI_XMT_DATA16 (TWI FIFO transmit data double byte) register, 23-49
TWI_XMT_DATA8 (TWI FIFO transmit data single byte) register, 23-48
two-dimensional DMA, 7-19
two-wire interface, *See* TWI
TX_ACT_MASK (data transmit active) bit, 27-67
TX_ACT (transmit active) bit, 27-64
TXCOL bit, 22-48
TXCOL flag, 22-25
TXCOL (transmit collision error) bit, 22-48

TX_COUNT (USB Tx byte count) bits, 26-133
TX_DAT_RDY_MASK (transmit data available) bit, 27-68
TX_DAT_RDY (transmit data available) bit, 27-63
TXE bit, 22-25, 22-48
TXECNT[7:0] field, 31-84
TXE (transmission error) bit, 22-48
TXF bit, 24-63, 24-66, 24-67
TxferSize (transfer size), 26-31, 26-34
TX_FIFO_FULL_MASK (transmit FIFO full) bit, 27-68
TX_FIFO_FULL (transmit FIFO full) bit, 27-63
TX_FIFO_STAT_MASK (transmit FIFO watermark) bit, 27-67
TX_FIFO_STAT (transmit FIFO watermark) bit, 27-64
TX_FIFO_ZERO_MASK (transmit FIFO empty) bit, 27-68
TX_FIFO_ZERO (transmit FIFO empty) bit, 27-63
TX hold register, 24-62
TXHRE bit, 24-67
TX_INTx_R (route Tx IRQ to INTx) bits, 26-104
TxPktRdy autoset enable (AUTOSET_T) bit, 26-117
TXPKTRDY (data packet in FIFO indicator) bit, 26-113
TXPKTRDY_T (data packet in FIFO indicator) bit, 26-117
TX_POLL_INTERVAL (USB Tx poll interval) bits, 26-130
Tx protocol type (PROTOCOL_T) bits, 26-129
TXREQ signal, 25-7
TXS bit, 22-31, 22-48
TXSE bit, 24-52, 24-56
TXS (SPIx_TDBR data buffer status) bit, 22-48
TX_UNDERRUN_MASK (transmit FIFO underrun error) bit, 27-67
TX_UNDERRUN_STAT (clear transmit FIFO underrun error) bit, 27-65
TX_UNDERRUN (transmit FIFO underrun error) bit, 27-64
TZAH (select minimum delay required for output) bits, 21-68

U

UART, 25-1 to 25-60
  autobaud detection, 10-33, 25-20, 25-54
  baud rate, 25-8
  baud rate examples, 25-19
  bit rate detection, 10-5
  bit rate examples, 25-19
  bitstream, 25-6
  block diagram, 25-3, 25-11
  booting, 25-20
  character transmission, 25-55
  clock, 25-18
  code examples, 25-51
  data words, 25-6
  divisor reset, 25-49
  DMA channels, 25-24
  DMA mode, 25-24
  errors during reception, 25-9
  external interfaces, 25-3
  features, 25-2
  glitch filtering, 25-14
  initialization, 25-51
  internal interfaces, 25-5
  interrupt channels, 25-41
  interrupt conditions, 25-44
  interrupts, 25-15
  IrDA mode, 25-2
  IrDA receiver, 25-14
  IrDA receiver pulse detection, 25-15

UART *(continued)*
IrDA transmit pulse, 25-13
IrDA transmitter, 25-13
and ISRs, 25-23
loopback mode, 25-32
mixing modes, 25-26
non-DMA interrupt operation, 25-57
non-DMA mode, 25-22
receive operation, 25-8
receive sampling window, 25-14
registers, table, 25-27
signals, 25-4
standard, 25-1
string transmission, 25-56
switching from DMA to non-DMA, 25-26
switching from non-DMA to DMA, 25-26
and system DMA, 25-41
transmission, 25-7
transmission SYNC bit use, 25-58
UART divisor latch high byte (UARTx_DLH) registers, 25-46
UART divisor latch low byte (UARTx_DLL) registers, 25-46
UART global control (UARTx_GCTL) registers, 25-50
UART interrupt enable clear (UARTx_IER_CLEAR) registers, 25-41
UART interrupt enable registers (UARTx_IER), 25-43
UART interrupt enable set (UARTx_IER_SET) registers, 25-41
UART interrupt enable (UARTx_IER) registers, 25-40
UART line control registers (UARTx_LCR), 25-29
UART line control (UARTx_LCR) registers, 25-29
UART line status registers (UARTx_LSR), 25-34
UART line status (UARTx_LSR) registers, 25-34
UART modem control (UARTx_MCR) registers, 25-32
UART modem status (UARTx_MSR) registers, 25-37
UART receive buffer registers (UARTx_RBR), 25-8
UART receive buffer (UARTx_RBR) registers, 25-40
UART scratch registers (UARTx_SCR), 25-49
UART scratch (UARTx_SCR) registers, 25-49
UART transmit holding (UARTx_THR) registers, 25-39
UARTx_DLH (UART divisor latch high byte registers), 25-27
UARTx_DLH (UART divisor latch high byte) registers, 25-46
UARTx_DLL, 25-27
UARTx_DLL (UART divisor latch low byte registers), 25-27
UARTx_DLL (UART divisor latch low byte) registers, 25-46
UARTx_GCTL (UART global control registers), 25-27
UARTx_GCTL (UART global control) registers, 25-50
UARTx_IER_CLEAR (UART interrupt enable clear) registers, 25-41
UARTx_IER_SET (UART interrupt enable set) registers, 25-41
UARTx_IER (UART interrupt enable registers), 25-43
UARTx_IER (UART interrupt enable) registers, 25-40

UARTx_IIR (UART interrupt identification registers), 25-28
UARTx_LCR (UART line control registers), 25-27, 25-29
UARTx_LCR (UART line control) registers, 25-29
UARTx_LSR (UART line status registers), 25-27, 25-34
UARTx_LSR (UART line status) registers, 25-34
UARTx_MCR (UART modem control registers), 25-27
UARTx_MCR (UART modem control) registers, 25-32
UARTx_MSR (UART modem status) registers, 25-37
UARTx_RBR (UART receive buffer registers), 25-8, 25-28
UARTx_RBR (UART receive buffer) registers, 25-40
UARTx_SCR (UART scratch registers), 25-28, 25-49
UARTx_SCR (UART scratch) registers, 25-49
UARTx_THR (UART transmit holding registers), 25-7, 25-28
UARTx_THR (UART transmit holding) registers, 25-39
UCCNF[3:0] field, 31-28, 31-83
UCCNT[15:0] field, 31-84
UCCT bit, 31-83
UCE bit, 31-83
UCEIF bit, 31-26, 31-50
UCEIM bit, 31-26, 31-49
UCEIS bit, 31-26, 31-49
UCEN bit, 25-8, 25-18, 25-50, 25-51
UCIE (up count interrupt enable) bit, 13-28
UCII (up count interrupt identifier) bit, 13-28
UCRC[15:0] field, 31-84
UCRC bit, 31-83
UDMAIN_CSTATE (ultra DMA-In mode state machine current state) bits, 21-59
UDMAIN_DONE_INT (ultra-DMA in transfer done interrupt status) bit, 21-57
UDMAIN_DONE_MASK (ultra-DMA in transfer done interrupt mask) bit, 21-55
UDMAIN_FIFO_THRS (ultra DMA-IN FIFO threshold) bits, 21-49
UDMA_IN_FL (ultra DMA input FIFO level) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-67, 21-68
UDMAIN_TERM_INT (device terminate ultra-DMA-in transfer interrupt status) bit, 21-57
UDMAIN_TERM_MASK (device terminate ultra-DMA-in transfer interrupt mask) bit, 21-55
UDMAIN_TFRCNT (UDMA in transfer count) bits, 21-62
UDMAOUT_CSTATE (ATAPI IORDY line status) bits, 21-59
UDMAOUT_DONE_INT (ultra-DMA out transfer done interrupt status) bit, 21-57
UDMAOUT_DONE_MASK (ultra-DMA out transfer done interrupt mask) bit, 21-55
UDMAOUT_TERM_INT (device terminate ultra-DMA-out transfer interrupt status) bit, 21-57
UDMAOUT_TERM_MASK (device terminate ultra-DMA-out transfer interrupt mask) bit, 21-55

UDMAOUT_TFRCNT (UDMA out transfer count) bits, 21-63
UDMA_XFER_ON (ultra DMA transfer in progress) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
UDS_MOD (resampling mode) bit, 28-37
UIAIF bit, 31-27, 31-50
UIAIM bit, 31-27, 31-49
UIAIS bit, 31-27, 31-49
ultra DMA-IN FIFO threshold (UDMAIN_FIFO_THRS) bits, 21-49
ultra DMA-In mode state machine current state (UDMAIN_CSTATE) bits, 21-59
ultra DMA input FIFO level (UDMA_IN_FL) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-67, 21-68
ultra DMA input FIFO level (ULTRA_IN_FL) bits, 21-51
ultra-DMA in transfer done interrupt mask (UDMAIN_DONE_MASK) bit, 21-55
ultra-DMA in transfer done interrupt status (UDMAIN_DONE_INT) bit, 21-57
ultra-DMA out transfer done interrupt mask (UDMAOUT_DONE_MASK) bit, 21-55
ultra-DMA out transfer done interrupt status (UDMAOUT_DONE_INT) bit, 21-57
ultra DMA transfer in progress (UDMA_XFER_ON) bit, 21-51, 21-52, 21-53, 21-54, 21-58, 21-59, 21-60, 21-61, 21-62, 21-63, 21-64, 21-65, 21-66, 21-67, 21-68
ultra DMA transfer in progress (ULTRA_XFER_ON) bit, 21-51
ULTRA_IN_FL (ultra DMA input FIFO level) bits, 21-51
ULTRA_START (start ultra-DMA Op) bit, 21-49
ULTRA_XFER_ON (ultra DMA transfer in progress) bit, 21-51
UNDERRUN_T (no TxPktRdy for IN token) bit, 26-117
unframed/framed, serial data, 24-33
universal asynchronous receiver/transmitter, *See* UART
universal counter, CAN, 31-28
universal counter exceeded interrupt, CAN, 31-26
unused pins, 19-17
Upper PBS00 Half Page (PBS00H, Bits 63–32), 17-114
Upper PBS01 Half Page (PBS01H, Bits 15–0), 17-117
Upper PBS01 Half Page (PBS01H, Bits 63–16), 17-116
urgent DMA transfers, 7-54
USB_APHY_CNTRL2 (USB APHY control 2) register, 26-141
USB APHY control 2 (USB_APHY_CNTRL2) register, 26-141
USB common interrupts enable (USB_INTRUSBE) register, 26-110
USB common interrupts (USB_INTRUSB) register, 26-109
USB control/status EP0 (USB_CSR0) register, 26-113

USB_COUNT0 (USB received byte count in EP0 FIFO) register, 26-128
USB_CSR0 (USB control/status EP0) register, 26-113
USB DMA endpoint x interrupt (DMAx_INT) bits, 26-144
USB_DMA_INTERRUPT (USB DMA interrupt) register, 26-144
USB DMA interrupt (USB_DMA_INTERRUPT) register, 26-144
USB DMAx address high (USB_DMAxADDRHIGH) register, 26-147
USB DMAx address low (USB_DMAxADDRLOW) register, 26-146
USB_DMAxADDRHIGH (USB DMAx address high) register, 26-147
USB_DMAxADDRLOW (USB DMAx address low) register, 26-146
USB_DMAxCONTROL (USB DMAx control) registers, 26-145
USB DMAx control (USB_DMAxCONTROL) registers, 26-145
USB_DMAxCOUNTHIGH (USB DMAx count high) register, 26-148
USB DMAx count high (USB_DMAxCOUNTHIGH) register, 26-148
USB_DMAxCOUNTLOW (USB DMAx count low) register, 26-147
USB DMAx count low (USB_DMAxCOUNTLOW) register, 26-147
USB enable (GLOBAL_ENA) bit, 26-98
USB endpoint index (SELECTED_ENDPOINT) bits, 26-103, 26-111
USB_EP_NI0_RXCOUNT register, A-46
USB_EP_NI0_RXCSR register, A-46
USB_EP_NI0_RXINTERVAL register, A-46
USB_EP_NI0_RXMAXP register, A-46
USB_EP_NI0_RXTYPE register, A-46
USB_EP_NI0_TXCOUNT register, A-47
USB_EP_NI0_TXCSR register, A-46
USB_EP_NI0_TXINTERVAL register, A-46
USB_EP_NI0_TXMAXP register, A-46
USB_EP_NI0_TXTYPE register, A-46
USB_EP_NI1_RXCOUNT register, A-47
USB_EP_NI1_RXCSR register, A-47
USB_EP_NI1_RXINTERVAL register, A-47
USB_EP_NI1_RXMAXP register, A-47
USB_EP_NI1_RXTYPE register, A-47
USB_EP_NI1_TXCOUNT register, A-48
USB_EP_NI1_TXCSR register, A-47
USB_EP_NI1_TXINTERVAL register, A-47
USB_EP_NI1_TXMAXP register, A-47
USB_EP_NI1_TXTYPE register, A-47
USB_EP_NI2_RXCOUNT register, A-48
USB_EP_NI2_RXCSR register, A-48
USB_EP_NI2_RXINTERVAL register, A-48
USB_EP_NI2_RXMAXP register, A-48
USB_EP_NI2_RXTYPE register, A-48
USB_EP_NI2_TXCOUNT register, A-49
USB_EP_NI2_TXCSR register, A-48
USB_EP_NI2_TXINTERVAL register, A-48
USB_EP_NI2_TXMAXP register, A-48
USB_EP_NI2_TXTYPE register, A-48
USB_EP_NI3_RXCOUNT register, A-49
USB_EP_NI3_RXCSR register, A-49
USB_EP_NI3_RXINTERVAL register, A-49

USB_EP_NI3_RXMAXP register, A-49
USB_EP_NI3_RXTYPE register, A-49
USB_EP_NI3_TXCOUNT register, A-50
USB_EP_NI3_TXCSR register, A-49
USB_EP_NI3_TXINTERVAL register, A-49
USB_EP_NI3_TXMAXP register, A-49
USB_EP_NI3_TXTYPE register, A-49
USB_EP_NI4_RXCOUNT register, A-50
USB_EP_NI4_RXCSR register, A-50
USB_EP_NI4_RXINTERVAL register, A-50
USB_EP_NI4_RXMAXP register, A-50
USB_EP_NI4_RXTYPE register, A-50
USB_EP_NI4_TXCOUNT register, A-51
USB_EP_NI4_TXCSR register, A-50
USB_EP_NI4_TXINTERVAL register, A-50
USB_EP_NI4_TXMAXP register, A-50
USB_EP_NI4_TXTYPE register, A-50
USB_EP_NI5_RXCOUNT register, A-51
USB_EP_NI5_RXCSR register, A-51
USB_EP_NI5_RXINTERVAL register, A-51
USB_EP_NI5_RXMAXP register, A-51
USB_EP_NI5_RXTYPE register, A-51
USB_EP_NI5_TXCOUNT register, A-52
USB_EP_NI5_TXCSR register, A-51
USB_EP_NI5_TXINTERVAL register, A-51
USB_EP_NI5_TXMAXP register, A-51
USB_EP_NI5_TXTYPE register, A-51
USB_EP_NI6_RXCOUNT register, A-52
USB_EP_NI6_RXCSR register, A-52
USB_EP_NI6_RXINTERVAL register, A-52
USB_EP_NI6_RXMAXP register, A-52
USB_EP_NI6_RXTYPE register, A-52
USB_EP_NI6_TXCOUNT register, A-53
USB_EP_NI6_TXCSR register, A-52
USB_EP_NI6_TXINTERVAL register, A-52
USB_EP_NI6_TXMAXP register, A-52
USB_EP_NI6_TXTYPE register, A-52
USB_EP_NI7_RXCOUNT register, A-53
USB_EP_NI7_RXCSR register, A-53
USB_EP_NI7_RXINTERVAL register, A-53
USB_EP_NI7_RXMAXP register, A-53
USB_EP_NI7_RXTYPE register, A-53
USB_EP_NI7_TXCOUNT register, A-53
USB_EP_NI7_TXCSR register, A-53
USB_EP_NI7_TXINTERVAL register, A-53
USB_EP_NI7_TXMAXP register, A-53
USB_EP_NI7_TXTYPE register, A-53
USB_FADDR (USB function address) register, 26-102
USB frame number (FRAME_NUMBER) bits, 26-111
USB frame number (USB_FRAME) register, 26-111
USB_FRAME (USB frame number) register, 26-111
USB_FS_EOF1 (USB full-speed EOF 1) register, 26-140
USB full-speed EOF 1 (USB_FS_EOF1) register, 26-140
USB function address (USB_FADDR) register, 26-102
USB global control (USB_GLOBAL_CTL) register, 26-98
USB_GLOBAL_CTL (USB global control) register, 26-98
USB global interrupt (USB_GLOBINTR) register, 26-104
USB_GLOBINTR (USB global interrupt) register, 26-104

USB hibernate signal (CSR_HBR) bit, 26-141
USB high-speed EOF 1 (USB_HS_EOF1) register, 26-139
USB_HS_EOF1 (USB high-speed EOF 1) register, 26-139
USB_INDEX (USB index) register, 26-103, 26-111
USB index (USB_INDEX) register, 26-103, 26-111
USB_INTRRXE (USB receive interrupt enable) register, 26-108
USB_INTRRX (USB receive interrupt) register, 26-106
USB_INTRTXE (USB transmit interrupt enable) register, 26-107
USB_INTRTX (USB transmit interrupt) register, 26-105
USB_INTRUSBE (USB common interrupts enable) register, 26-110
USB_INTRUSB (USB common interrupts) register, 26-109
USB_INTx_R (route USB/VBUS IRQ to INTx) bits, 26-104
USB_LINKINFO (USB link info) register, 26-138
USB link info (USB_LINKINFO) register, 26-138
USB low-speed EOF 1 (USB_LS_EOF1) register, 26-140
USB_LS_EOF1 (USB low-speed EOF 1) register, 26-140
USB max Rx data in frame (MAX_PACKET_SIZE_R) bits, 26-122
USB max Tx data in frame (MAX_PACKET_SIZE_T) bits, 26-112
USB_NAKLIMIT0 (USB NAK limit 0) register, 26-130
USB NAK limit 0 (USB_NAKLIMIT0) register, 26-130
USB or non-USB part (USBPARTB1V) bit, 26-141
USB OTG
  DMA master channels, 26-90
  features, 26-2
  host negotiation /configuration, 26-82
  interface pins, 26-55
  OTG session request, 26-80
  peripheral mode operation, 26-13
  transferring packets using DMA, 26-92
USB_OTG_DEV_CTL (USB OTG device control) register, 26-134, 26-136
USB OTG device control (USB_OTG_DEV_CTL) register, 26-134, 26-136
USB_OTG_VBUS_MASK (USB OTG VBUS mask) register, 26-138
USB OTG VBUS mask (USB_OTG_VBUS_MASK) register, 26-138
USBPARTB1V (USB part or non-USB part) bit, 26-141
USB peripheral device address (FUNCTION_ADDRESS) bits, 26-102
USB PLL OSC control (USB_PLLOSC_CTRL) register, 26-142
USB_PLLOSC_CTRL (USB PLL OSC control) register, 26-142
USB power management (USB_POWER) register, 26-99
USB_POWER (USB power management) register, 26-99
USB pu/pd restore control (CSR_RSTD) bit, 26-141

USB received byte count in EP0 FIFO (USB_COUNT0) register, 26-128
USB receive interrupt enable (USB_INTRRXE) register, 26-108
USB receive interrupt (USB_INTRRX) register, 26-106
USB reset (RESET) bit, 26-99
USB Rx byte count (RX_COUNT) bits, 26-129
USB Rx byte count (USB_RXCOUNT) register, 26-129
USB Rx control/status EPx (USB_RXCSR) register, 26-123
USB_RXCOUNT (USB Rx byte count) register, 26-129
USB_RXCSR (USB Rx control/status EPx) register, 26-123
USB Rx endpoint x interrupt enable (EPx_RX_E) bits, 26-108
USB Rx endpoint x interrupt (EPx_RX) bits, 26-106
USB_RXINTERVAL (USB Rx interval) register, 26-132
USB Rx interval (USB_RXINTERVAL) register, 26-132
USB_RX_MAX_PACKET (USB Rx max packet) register, 26-122
USB Rx max packet (USB_RX_MAX_PACKET) register, 26-122
USB Rx poll interval (RX_POLL_INTERVAL) bits, 26-132
USB_RXTYPE (USB Rx type) register, 26-131
USB Rx type (USB_RXTYPE) register, 26-131
USB_SRP_CLKDIV (USB SRP clock divider) register, 26-143
USB SRP clock divider (USB_SRP_CLKDIV) register, 26-143
USB transmit interrupt enable (USB_INTRTXE) register, 26-107
USB transmit interrupt (USB_INTRTX) register, 26-105
USB Tx byte count (TX_COUNT) bits, 26-133
USB Tx byte count (USB_TXCOUNT) register, 26-133
USB Tx control/status EPx (USB_TXCSR) register, 26-117
USB_TXCOUNT (USB Tx byte count) register, 26-133
USB_TXCSR (USB Tx control/status EPx) register, 26-117
USB Tx endpoint x interrupt enable (EPx_TX_E) bits, 26-107
USB Tx endpoint x interrupt (EPx_TX) bits, 26-105
USB_TXINTERVAL (USB Tx interval) register, 26-130
USB Tx interval (USB_TXINTERVAL) register, 26-130
USB_TX_MAX_PACKET (USB Tx max packet) register, 26-112
USB Tx max packet (USB_TX_MAX_PACKET) register, 26-112
USB Tx poll interval (TX_POLL_INTERVAL) bits, 26-130
USB_TXTYPE (USB Tx type) register, 26-129
USB Tx type (USB_TXTYPE) register, 26-129
USB VBUS pulse length (USB_VPLEN) register, 26-139

USB_VPLEN (USB VBUS pulse length) register, 26-139
user mode, 17-10
UTE bit, 7-49, 7-113
UTHE[15:0] field, 7-117

V

Valid bit
  clearing, 3-41
  figure, 3-26
  function, 3-16
  in cache-line replacement, 3-18
  in instruction cache invalidation, 3-22
valid (definition), 3-81
VBUS1–0 (VBUS level indicator) bit, 26-134, 26-136
VBUS_ERROR_BE (VBus threshold IRQ enable) bit, 26-110
VBUS_ERROR_B (VBus threshold indicator) bit, 26-109
VBUS level indicator (VBUS1–0) bit, 26-134, 26-136
VBUS pulse length (VPLEN) bits, 26-139
VBus threshold indicator (VBUS_ERROR_B) bit, 26-109
VBus threshold IRQ enable (VBUS_ERROR_BE) bit, 26-110
VCO, multiplication factors, 18-4
victim (definition), 3-81
VLEV[3:0] field, 18-19, 18-28
voltage, 18-16
  changing, 18-19
  control, 18-7
  dynamic control, 18-16
voltage controlled oscillator (VCO), 18-3
voltage level values, 18-19
voltage regulator, 1-33
voltage regulator control register (VR_CTL), 18-17, 18-28
VPLEN (VBUS pulse length) bits, 26-139
VR_CTL (voltage regulator control register), 18-17, 18-25, 18-28, 31-40

W

wait for connect (WTCON) bits, 26-138
wait for response (CMD_RSP) bit, 27-58
wait from IDPULLUP (WTID) bits, 26-138
WAKE bit, 18-28
wakeup function, 6-14
wakeup interrupt, CAN, 31-27
Wakeup Preamble Detected interrupt enable, 29-45
Wake-Up Preamble Received (WUP) interrupt event, 29-34
Wake-Up (WAKEUP) bit, 29-18
watchdog control register (WDOG_CTL), 12-8, 12-9
watchdog count[15:0] field, 12-6
watchdog count[31:16] field, 12-6
watchdog count register (WDOG_CNT), 12-6
watchdog mode, CAN, 31-20
watchdog status[15:0] field, 12-7
watchdog status[31:16] field, 12-7
watchdog status register (WDOG_STAT), 12-7
watchdog timer, 1-30, 12-1 to 12-11
  block diagram, 12-3
  disabling, 12-5
  and emulation mode, 12-2
  features, 12-1
  registers, 12-6
  and reset, 12-5
  reset, 17-5, 17-7
  starting, 12-4
  zero value, 12-5
watermark level (WM_LVL) bits, 28-37
waveform generation, pulse width modulation, 10-16

Way
1-Way associative (direct-mapped), 3-79
definition, 3-81
L1 instruction memory as 4-Way set-associative, 3-6
priority in cache-line replacement, 3-19
WBA bit, 31-45
WB_EDGE (write buffer edge detect) bit, 20-20, 20-21
WB_FULL (write buffer full) bit, 20-20
WB_OVF (write buffer overflow) bit, 20-21
WDEN[7:0] field, 12-8
WDEV[1:0] field, 12-4, 12-8
WDOG_CNT (watchdog count register), 12-4, 12-6
WDOG_CTL (watchdog control register), 12-8, 12-9
WDOG_STAT (watchdog status register), 12-4, 12-7
WDRESET, 16-58, 16-59, 17-106
WDSIZE[1:0] field, 7-80, 7-84
WDTH_CAP mode, 10-25, 10-47
control bit and register usage, 10-51
WLS[1:0] field, 25-29
WM_LVL (watermark level) bits, 28-37
WNR bit, 7-80, 7-84, 8-6
WOFF[9:0] field, 24-24, 24-69
WOM bit, 22-20, 22-45
WOM (write open drain master) bit, 22-45
word length
SPI, 22-21
SPORT, 24-30
SPORT receive data, 24-65
SPORT transmit data, 24-62
work unit
completion, 7-30
DMA, 7-21
interrupt timing, 7-33
restrictions, 7-32
work unit *(continued)*
transitions, 7-32
WR bit, 31-46
WR_DLY (write strobe delay) bits, 20-19
WR_DONE (page write done) bit, 20-21
write, 3-81
write back (definition), 3-81
write buffer depth, 3-40
write buffer edge detect (WB_EDGE) bit, 20-20, 20-21
write buffer full (WB_FULL) bit, 20-20
write buffer overflow (WB_OVF) bit, 20-21
write complete interrupt enable bit, 14-21
write open drain master (WOM) bit, 22-45
write pending status bit, 14-22, 14-23
write strobe delay (WR_DLY) bits, 20-19
WSIZE[3:0] field, 24-23, 24-69
WT bit, 31-46
WTCON (wait for connect) bits, 26-138
WTID (wait from IDPULLUP) bits, 26-138
WUIF bit, 31-27, 31-50
WUIM bit, 31-27, 31-49
WUIS bit, 31-27, 31-49
WUPEN, 29-45
WURESET, 16-58, 16-59, 17-106

X

X_COUNT[15:0] field, 7-93
XFER_DIR (transfer direction) bit, 21-49
XFER_LENGTH (transfer length) bits, 21-58
XFR_TYPE (operating mode) bits, 15-81
X_MODIFY[15:0] field, 7-97
XMTDATA16[15:0] field, 23-49
XMTDATA8[7:0] field, 23-48
XMTFLUSH bit, 23-39, 23-40
XMTINTLEN bit, 23-39, 23-40
XMTSERV bit, 23-44, 23-46

XMTSERVM bit, 23-43
XMTSTAT[1:0] field, 23-41, 23-42
XOFF (transmitter off) bit, 25-32

Y

Y_COUNT[15:0] field, 7-99
YFIFO_ERR (luma FIFO error) bit, 15-80
YFIFO_ERR (Luma FIFO Overflow Error) bit, 21-49, 21-55, 21-56
Y_MODIFY[15:0] field, 7-104

Z

ZMZC (CZM zeroes counter enable) bit, 13-27
µ-law companding, 24-26, 24-31