

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

Revision 0.4, August 2008

Part Number
82-000001-02

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



Copyright Information

© 2008 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Lockbox Secure Technology Disclaimer

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

Trademark and Service Mark Notice

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

Lockbox is a trademark of Analog Devices, Inc.

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

CONTENTS

PREFACE

Contents of Two Volumes	xli
Purpose of This Manual	xlii
Intended Audience	xlii
Manual Contents	xliii
What's New in This Manual	xlvi
Technical or Customer Support	xlvi
Supported Processors	xlvii
Conventions	xlviii
Register Diagram Conventions	xlvi

INTRODUCTION

Peripherals	20-3
Memory Architecture	20-6
Internal Memory	20-7
External Memory	20-8
NAND Flash Controller (NFC)	20-9

I/O Memory Space	20-10
One-Time-Programmable (OTP) Memory	20-10
DMA Support	20-11
Host DMA Interface	20-13
External Bus Interface Unit	20-14
DDR SDRAM Controller	20-14
Asynchronous Controller	20-15
Ports	20-15
General-Purpose I/O (GPIO)	20-15
Two-Wire Interfaces	20-16
Controller Area Network	20-17
Enhanced Parallel Peripheral Interface (EPPI)	20-18
SPORT Controllers	20-20
Serial Peripheral Interface (SPI) Ports	20-22
Timers	20-22
UART Ports	20-23
USB On-The-Go, Dual-Role Device Controller	20-24
ATA/ATAPI-6 Interface	20-25
Keypad Interface	20-25
Secure Digital (SD)/SDIO Controller	20-26
Rotary Counter and Thumbwheel Interface	20-27
Security	20-27
Media Transceiver (MXVR) MAC Layer	20-29
Real-Time Clock	20-30

Contents

Watchdog Timer	20-31
Clock Signals	20-32
Dynamic Power Management	20-32
Full On Mode (Maximum Performance)	20-33
Active Mode (Moderate Dynamic Power Savings)	20-33
Sleep Mode (High Dynamic Power Savings)	20-33
Deep Sleep Mode (Maximum Dynamic Power Savings)	20-34
Hibernate State (Maximum Power Savings)	20-34
Voltage Regulation	20-34
Boot Modes	20-35
Instruction Set Description	20-35
Development Tools	20-36

MEDIA TRANSCEIVER MODULE (MXVR)

Overview	21-1
Interface Signals	21-3
MXVR Memory Map	21-5
MXVR Registers	21-6
MXVR Configuration Register (MXVR_CONFIG)	21-12
MXVR State Registers	21-19
MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0) .	21-29
MXVR Interrupt Status Register_1 (MXVR_INT_STAT_1)	21-40
MXVR Interrupt Enable Register 0 (MXVR_INT_EN_0) ...	21-43
MXVR Interrupt Enable Register 1 (MXVR_INT_EN_1) ...	21-46
MXVR Node Position Register (MXVR_POSITION)	21-48

MXVR Maximum Node Position Register	21-50
MXVR Node Frame Delay Register (MXVR_DELAY)	21-51
MXVR Maximum Node Frame Delay Register (MXVR_MAX_DELAY)	21-53
MXVR Logical Address Register (MXVR_LADDR)	21-54
MXVR Group Address Register (MXVR_GADDR)	21-55
MXVR Alternate Address Register (MXVR_AADDR)	21-56
MXVR Allocation Table Registers	21-56
MXVR Synchronous Logical Channel Assignment Registers .	21-58
MXVR DMAx Configuration Registers	21-60
MXVR DMA Channel x Start Address Registers	21-70
MXVR DMA Channel x Current Address Registers	21-72
MXVR DMA Channel x Transfer Count Registers	21-73
MXVR DMA Channel x Current Transfer Count Registers ...	21-76
MXVR Asynchronous Packet Control Register (MXVR_AP_CTL)	21-77
MXVR Asynch Packet Receive Buffer Start Address Register	21-80
MXVR Asynch Packet Receive Buffer Current Address	21-81
MXVR Asynch Packet Transmit Buffer Start Address Register	21-82
MXVR Asynch Packet Transmit Buffer Current Address	21-83
MXVR Control Message Control Register.....	21-83
MXVR Control Message Receive Buffer Start Address	21-86
MXVR Control Message Receive Buffer Current Address	21-87
MXVR Control Message Transmit Buffer Start Address	21-88
MXVR Control Message Transmit Buffer Current Address ..	21-89

Contents

MXVR Remote Read Buffer Start Address Register	21-90
MXVR Remote Read Buffer Current Address Register	21-91
MXVR Pattern Registers	21-91
MXVR Pattern Data Registers	21-92
MXVR Pattern Enable Registers	21-93
MXVR Frame Counter Registers	21-94
MXVR Routing Registers	21-95
MXVR Block Counter Register (MXVR_BLOCK_CNT)	21-98
MXVR Clock Control Register (MXVR_CLK_CTL)	21-100
MXVR Clock/Data Recovery PLL Control Register	21-107
MXVR Frequency Multiply PLL Control Register)	21-110
MXVR Pin Control Register (MXVR_PIN_CTL)	21-112
MXVR System Clock Counter Register	21-113
General Operation	21-115
Network Services Software	21-115
Network Activity Detection	21-115
Node Initialization	21-117
Initialization of Processor Pin Multiplexing	21-118
Master mode initialization, MXVR_CONFIG register ...	21-118
Slave mode initialization, MXVR_CONFIG register	21-118
Initialization of the MXVR_CLK_CTL register	21-119
Initialization of the MXVR_ROUTING_x registers	21-119
Initialization of the buffer start address registers	21-120
Enabling the MXVR PLLs	21-120

Enabling MXVR Output Clocks	21-120
Network Lock	21-121
Network Initialization	21-121
Synchronous Data Routing, Muting, and Transmission	21-123
Synchronous Data Reception	21-126
Asynchronous Packet Transmission	21-126
Asynchronous Packet Reception	21-129
Control Message Transmission	21-131
Normal Control Message Transmission	21-135
Remote Read Control Message Transmission	21-137
Remote Write Control Message Transmission	21-139
Resource Allocate Control Message Transmission	21-141
Resource De-Allocate Control Message Transmission	21-144
Remote Get Source Control Message Transmission	21-147
Control Message Reception	21-150
Normal Control Message Reception	21-151
Remote Read and Remote Write Reception	21-153
Resource Allocate Reception	21-154
Resource De-Allocate Reception	21-155
Remote Get Source Reception	21-156
MXVR Low Power Operation	21-156
Full On Mode	21-158
Active Mode	21-159
Sleep Mode	21-160

Contents

Deep Sleep Mode	21-162
Hibernate State	21-163
Power Gating the ADSP-BF54x	21-164

KEYPAD INTERFACE

Interface Overview	22-1
Description of Operation	22-2
Keypad Operation	22-2
Keypad Enable/Disable	22-4
Input Keypad Matrix Programmability	22-4
Waking Up on Keypad Press	22-4
Sensitivity of Keypad Interface	22-5
Limited Multiple Key Resolution	22-5
Keypad Interrupt Modes	22-6
Implementing Press-Hold Feature	22-6
Functional Description	22-7
State Diagram	22-7
Programming Model	22-9
Keypad Registers	22-10
Keypad Control Register (KPAD_CTL)	22-10
KPAD_PRESCALE Register	22-14
KPAD_MSEL Register	22-15
KPAD_ROWCOL Register	22-16
KPAD_STAT Register	22-19
KPAD_SOFTEVAL Register	22-21

Programming Examples 22-22

SECURE DIGITAL HOST

Overview 23-1

Interface Overview 23-2

Description of Operation 23-3

Functional Description 23-4

 SDH Clocking 23-4

 SDH Operation 23-5

 SDH Data 23-10

 WAIT_R 23-11

 RECEIVE 23-11

 SEND 23-12

 SDH Data FIFO 23-16

 Transmit FIFO 23-16

 Receive FIFO 23-17

 SDIO Interrupt and Read Wait Support 23-17

 MMC/SD Card Detection 23-18

 SDH DMA Transfers 23-19

Programming Model 23-19

SDH Registers 23-19

 SDH Power Control Register (SDH_PWR_CTL) 23-22

 SDH Clock Control Register (SDH_CLK_CTL) 23-23

 SDH Argument Register (SDH_ARGUMENT) 23-24

 SDH Command Register (SDH_COMMAND) 23-24

Contents

SDH Response Command Register (SDH_RESP_CMD)	23-25
SDH Response Registers (SDH_RESPONSEx)	23-26
SDH Data Timer Register (SDH_DATA_TIMER)	23-26
SDH Data Length Register (SDH_DATA_LGTH)	23-27
SDH Data Control Register (SDH_DATA_CTL)	23-27
SDH Data Counter Register (SDH_DATA_CNT)	23-29
SDH Status Register (SDH_STATUS)	23-30
SDH Status Clear Register (SDH_STATUS_CLR)	23-32
SDH Interrupt Mask Registers (SDH_MASKx)	23-33
SDH FIFO Counter Register (SDH_FIFO_CNT)	23-34
SDH Data FIFO (SDH_FIFOx) Registers	23-35
SDH Exception Status Register (SDH_E_STATUS)	23-35
SDH Exception Mask Register (SDH_E_MASK)	23-35
SDH Configuration Register (SDH_CFG)	23-36
SDH Read Wait Enable Register (SDH_RD_WAIT_EN)	23-38
SDH Identification Registers (SDH_PIDx)	23-38
Programming Examples	23-39
ATAPI INTERFACE	
Interface Overview	24-1
Description of Operation	24-4
Host PIO/Register Transfers	24-4
PIO Data-Out Transfers (Device Write)	24-6
PIO Data-In Transfers (Device Read)	24-8
Host Multiword DMA Transfers	24-11

Host Pausing the Multi-DMA Transfer	24-14
Host Terminating the Multi DMA Transfer	24-14
Device Pausing the Multi-DMA Transfer	24-14
Device Terminating the Multi-DMA Transfer	24-15
Host Ultra DMA Command Protocol Transfers	24-16
Host Pausing the Ultra DMA Data-In Transfer	24-17
Host Terminating the Ultra DMA Data-In Transfer	24-17
Device Pausing the Ultra DMA Data-In Transfer	24-17
Device Terminating the Ultra DMA Data-In Transfer	24-18
Host Pausing Ultra DMA Data-Out Transfer	24-18
Host Terminating Ultra DMA Data-Out Transfer	24-18
Device Pausing the Ultra DMA Data-Out Transfer	24-18
Device Terminating the Ultra DMA Data-Out Transfer ...	24-18
Functional Description	24-19
Power-on and Hardware Reset Protocol	24-19
Device Selection Protocol	24-21
Programmed I/O (PIO)	24-22
Host Multi DMA Block Implementation	24-23
Host Ultra DMA Block Implementation	24-29
Initiating an Ultra DMA Data-In Burst	24-29
Data-In Transfer	24-32
Device pausing an Ultra DMA Data-In Burst	24-33
Host pausing an Ultra DMA Data-In Burst	24-33
Ultra DMA Timing	24-35

Contents

Ultra DMA-Out Timing	24-39
Programming Model	24-43
ATAPI Device Configuration and Setup	24-43
PIO Data-out Transfers Pseudo-code	24-45
Host Multiword DMA Transfers Pseudo-code	24-46
Host Ultra DMA Command Protocol Transfers Pseudo-code	24-47
ATAPI Registers	24-48
ATAPI Control and Status Registers	24-49
ATAPI Control Register (ATAPI_CONTROL)	24-50
ATAPI Status Register (ATAPI_STATUS)	24-52
ATAPI Device Address Register (ATAPI_DEV_ADDR) ...	24-53
ATAPI Device Transmit Buffer Register	24-54
ATAPI Device Receive Buffer Register	24-55
ATAPI Interrupt Mask (ATAPI_INT_MASK) Register	24-56
ATAPI Interrupt Status Register (ATAPI_INT_STATUS)	24-57
ATAPI Transfer Length Register (ATAPI_XFER_LEN)	24-59
ATAPI Line Status Register (ATAPI_LINE_STATUS)	24-60
ATAPI State Machine Status Register	24-61
ATAPI Host Terminate Register (ATAPI_TERMINATE) .	24-61
ATAPI PIO Transfer Count Register	24-62
ATAPI Multiword DMA Transfer Count	24-62
ATAPI Ultra DMA Transfer Count	24-63
ATAPI Ultra DMA OUT Transfer Count	24-64
ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0)	24-64

ATAPI Programmed I/O Timing 0	24-65
ATAPI Programmed I/O Timing 1	24-65
ATAPI Multi DMA Timing 0	24-66
ATAPI Multi DMA Timing 1	24-66
ATAPI Multi DMA Timing 2	24-67
ATAPI Ultra DMA Timing 0	24-67
ATAPI Ultra DMA Timing 1	24-68
ATAPI Ultra DMA Timing 2 Register	24-68
ATAPI Ultra DMA Timing 3 Register	24-69
ATAPI Device I/O Registers	24-69
Command Register (R/W)	24-71
Device Control Register (WO)	24-71
Features Register (WO)	24-72
Sector Count Register (R/W)	24-72
Status Register (RO)	24-72
Alternate Status Register (RO)	24-73
Error Register (RO)	24-73
ATAPI Standards Reference	24-74
Summary of IDE/ATA Standards	24-78
ATAPI Timing Summary	24-79
IDE/ATA Transfer Modes and Protocols	24-79
Programmed (I/O) PIO Modes	24-79
Direct Memory Access (DMA) Modes	24-80
Ultra Direct Memory Access (DMA) Modes	24-80

Contents

ATAPI Device Selection	24-81
------------------------------	-------

NAND FLASH CONTROLLER

Overview	25-2
Interface Overview	25-4
Description of Operation	25-5
Internal Bus Interfaces	25-5
Bus Access Types	25-6
Access Timing	25-6
Pin Sharing	25-7
Functional Description	25-7
Page Write	25-8
Page Read	25-9
Additional Operations	25-10
Write Protection	25-11
Chip Enable Don't Care	25-11
NFC Error Detection	25-11
Error Analysis	25-13
Large Page Size Support	25-15
NFC SmartMedia Support	25-15
Programming Model	25-15
NFC Registers	25-17
NFC Control Register (NFC_CTL)	25-19
NFC Status Register (NFC_STAT)	25-20
NFC Interrupt Status Register (NFC_IRQSTAT)	25-21

NFC Interrupt Mask Register (NFC_IRQMASK)	25-23
NFC ECC Registers (NFC_ECCx)	25-23
NFC Count Register (NFC_COUNT)	25-25
NFC Reset Register (NFC_RST)	25-25
NFC Page Control Register (NFC_PGCTL)	25-26
NFC Read Data Register (NFC_READ)	25-26
NFC Address Register (NFC_ADDR)	25-27
NFC Command Register (NFC_CMD)	25-28
NFC Data Write Register (NFC_DATA_WR)	25-29
NFC Data Read Register (NFC_DATA_RD)	25-29
NFC Programming Examples	25-30

ENHANCED PARALLEL PERIPHERAL INTERFACE

Overview	26-1
Interface Overview	26-5
Description of Operation	26-7
EPPi Reset	26-8
Clock Gating	26-8
Frame Sync Polarity & Sampling Edge	26-9
Interrupts	26-10
Functional Description	26-11
ITU-R 656 Modes	26-11
ITU-R 656 Background	26-11
ITU-R 656 Input Modes	26-17
Entire Field	26-17

Contents

Active Video	26-18
Vertical Blanking Interval (VBI) only	26-18
ITU-R 656 Output in GP Transmit Modes	26-19
Frame Synchronization in ITU-R 656 Modes	26-22
General-Purpose EPPI Modes	26-23
GP 0 FS Mode	26-24
Frame Synchronization in GP 0 FS External Trigger Mode	26-25
Frame Synchronization in GP 0 FS Internal Trigger Mode	26-25
GP 1 FS Mode	26-25
GP 2 FS Mode	26-26
DEN functionality in GP 2 FS Transmit Mode	26-27
GP 3 FS Mode	26-28
EPPI Data Path Options	26-29
EPPI Data Lengths	26-29
EPPI DMA Channels	26-30
Data Packing For Receive Modes	26-30
Data Unpacking For Transmit Modes	26-31
Sign-Extension and Zero-Filling	26-32
Split Receive Modes	26-33
Split Transmit Modes	26-33
RGB Data Formats	26-34
Programmed Clipping and Thresholding of Data Values	26-34
Data Transfer Examples	26-35
8-Bit Receive Mode	26-35

10/12/14-Bit Receive Modes	26-37
16-Bit Receive Mode	26-40
18-Bit Receive Mode	26-42
24-Bit Receive Mode	26-44
8-Bit Split Receive Mode	26-45
10/12/14/16-Bit Split Receive Mode with SPLT_16 = 0 ...	26-48
16-Bit Split Receive Mode with SPLT_16 = 1	26-50
8-Bit Transmit Mode	26-51
10/12/14-Bit Transmit Modes	26-52
16-Bit Transmit Mode	26-53
18-Bit Transmit Mode	26-55
24-Bit Transmit Mode	26-56
8-Bit Split Transmit Mode	26-56
10/12/14/16-Bit Split Transmit Mode with SPLT_16 = 0 .	26-61
16-Bit Split Transmit Mode with SPLT_16 = 1	26-64
Programming Model	26-66
DMA Operation	26-66
Elevating EPPI Urgent requests at DDR controller Interface .	26-74
System Configuration	26-76
EPPI Registers	26-76
PPIx_CONTROL Register	26-79
PPIx_STATUS Register	26-86
Windowing Registers	26-90
EPPI Lines per Frame Register (PPIx_FRAME)	26-92

Contents

EPPI Samples per Line Register (PPIx_LINE)	26-92
EPPI Vertical Delay Register (PPIx_VDELAY)	26-93
EPPI Vertical Transfer Count Register (PPIx_VCOUNT)	26-93
EPPI Horizontal Delay Register (PPIx_HDELAY)	26-94
EPPI Horizontal Transfer Count Register	26-95
EPPI Clock Divide Register (PPIx_CLKDIV)	26-95
Frame Sync/ Blanking Generation Registers	26-96
EPPI FS1 Width Register / EPPI Horizontal Blanking Samples per Line Register (PPIx_FS1W_HBL)	26-96
EPPI FS2 Width Register/ EPPI Lines of Vertical Blanking Register (PPIx_FS2W_LVB)	26-96
EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (PPIx_FS1P_AVPL)	26-98
EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (PPIx_FS2P_LAVF)	26-99
EPPI Clipping Register (PPIx_CLIP)	26-101

CAN MODULE

Overview	27-1
Interface Overview	27-2
CAN Mailbox Area	27-5
CAN Mailbox Control	27-7
CAN Protocol Basics	27-8
CAN Operation	27-10
Bit Timing	27-11
Transmit Operation	27-13

Retransmission	27-14
Single Shot Transmission	27-15
Auto-Transmission	27-16
Receive Operation	27-16
Data Acceptance Filter	27-20
Watchdog Mode	27-21
Time Stamps	27-21
Remote Frame Handling	27-22
Temporarily Disabling Mailboxes	27-23
Functional Operation	27-25
CAN Interrupts	27-25
Mailbox Interrupts	27-25
Global CAN Interrupt	27-26
Event Counter	27-29
CAN Warnings and Errors	27-30
Programmable Warning Limits	27-30
CAN Error Handling	27-30
Error Frames	27-31
Error Levels	27-33
Debug and Test Modes	27-35
Low Power Features	27-39
CAN Built-In Suspend Mode	27-39
CAN Built-In Sleep Mode	27-40
CAN Wakeup From Hibernate State	27-40

Contents

CAN Registers	27-41
Global CAN Registers	27-46
CANx_CONTROL Master Control Registers	27-46
CANx_STATUS Global CAN Status Registers	27-47
CANx_DEBUG Registers	27-48
CANx_CLOCK Registers	27-48
CANx_TIMING Registers	27-49
CANx_INTR Interrupt Pending Registers	27-49
CANx_GIM Global CAN Interrupt Mask Registers	27-50
CANx_GIS Global CAN Interrupt Status Registers	27-51
CANx_GIF Global CAN Interrupt Flag Registers	27-52
Mailbox/Mask Registers	27-52
CANx_AMxx Acceptance Mask Registers	27-53
CANx_MBxx_ID1 Registers	27-57
CANx_MBxx_ID0 Registers	27-59
CANx_MBxx_TIMESTAMP Registers	27-61
CANx_MBxx_LENGTH Registers	27-63
CANx_MBxx_DATAx Registers	27-65
Mailbox Control Registers	27-72
CANx_MCx Mailbox Configuration Registers	27-73
CANx_MDx Mailbox Direction Registers	27-74
CANx_RMPx Registers	27-75
CANx_RMLx Registers	27-76
CANx_OPSSx Register	27-77

CANx_TRSx Registers	27-78
CANx_TRRx Registers	27-79
CANx_AAx Registers	27-80
CANx_TAx Registers	27-81
CANx_MBTD Register	27-82
CANx_RFHx Registers	27-83
CANx_MBIMx Registers	27-84
CANx_MBTIFx Registers	27-85
CANx_MBRIFx Registers	27-86
Universal Counter Registers	27-87
CANx_UCCNF Register	27-87
CANx_UCCNT Register	27-88
CANx_UCRC Register	27-88
Error Registers	27-89
CANx_CEC Register	27-89
CANx_ESR Register	27-89
CANx_EWR Register	27-90
Programming Examples	27-91
CAN Setup Code	27-91
Initializing and Enabling CAN Mailboxes	27-93
Initiating CAN Transfers and Processing Interrupts	27-94

SPI-COMPATIBLE PORT CONTROLLERS

Overview	28-1
Interface Overview	28-3

Contents

External Interface	28-4
Serial Peripheral Interface Clock Signal (SPIxSCK)	28-5
Master Out Slave In (MOSI)	28-6
Master In Slave Out (MISO)	28-6
Serial Peripheral Interface Slave Select Input Signal	28-7
Serial Peripheral Interface Slave Select Enable Output	28-9
Slave Select Inputs	28-12
Use of FLS Bits in SPI_FLG for Multiple Slave SPI	28-12
Internal Interfaces	28-14
DMA Functionality	28-14
SPI Transmit Data Buffer	28-15
SPI Receive Data Buffer	28-16
Description of Operation	28-16
SPI Transfer Protocols	28-17
SPI General Operation	28-19
SPI Control	28-21
Clock Signals	28-22
SPI Baud Rate	28-22
Error Signals and Flags	28-23
Mode Fault Error (MODF)	28-24
Transmission Error (TXE)	28-25
Reception Error (RBSY)	28-25
Transmit Collision Error (TXCOL)	28-25
Interrupt Output	28-25

Functional Description	28-26
Master Mode Operation	28-26
Transfer Initiation From Master (Transfer Modes)	28-28
Slave Mode Operation	28-29
Slave Ready for a Transfer	28-30
Programming Model	28-30
Beginning and Ending an SPI Transfer	28-30
Master Mode DMA Operation	28-33
Slave Mode DMA Operation	28-35
SPI Registers	28-43
SPI Baud Rate (SPIx_BAUD) Register	28-44
SPI Control (SPIx_CTL) Register	28-45
SPI Flag (SPIx_FLG) Register	28-46
SPI Status (SPIx_STAT) Register	28-48
SPI Transmit Data Buffer (SPIx_TDBR) Register	28-48
SPI Receive Data Buffer (SPIx_RDBR) Register	28-49
SPI RDBR Shadow (SPIx_SHADOW) Register	28-49
Programming Examples	28-50
Core Generated Transfer	28-50
Initialization Sequence	28-50
Starting a Transfer	28-51
Post Transfer and Next Transfer	28-52
Stopping	28-53
DMA Transfer	28-53

Contents

DMA Initialization Sequence	28-54
SPI Initialization Sequence	28-55
Starting a Transfer	28-56
Stopping a Transfer	28-56

TWO WIRE INTERFACE CONTROLLERS

Overview	29-1
Interface Overview	29-3
External Interface	29-4
Serial Clock signal (SCL1–0)	29-4
Serial data signal (SDA1–0)	29-5
TWI Pins	29-5
Internal Interfaces	29-6
Description of Operation	29-7
TWI Transfer Protocols	29-7
Clock Generation and Synchronization	29-7
Bus Arbitration	29-8
Start and Stop Conditions	29-9
General Call Support	29-10
Fast Mode	29-11
TWI General Operation	29-11
TWI Control	29-11
Clock Signal	29-12
Error Signals and Flags	29-13
TWI Master Status	29-13

TWI Slave Status	29-16
TWI FIFO Status	29-17
TWI Interrupt Status	29-18
Functional Description	29-22
General Setup	29-22
Slave Mode	29-22
Master Mode Clock Setup	29-24
Master Mode Transmit	29-24
Master Mode Receive	29-25
Clock Stretching	29-26
Repeated Start Condition	29-29
Programming Model	29-32
TWI Registers	29-34
TWIx_CONTROL Register	29-36
TWIx_CLKDIV Register	29-36
TWIx_SLAVE_CTL Register	29-37
TWIx_SLAVE_ADDR Register	29-39
TWIx_SLAVE_STAT Register	29-40
TWIx_MASTER_CTL Register	29-41
TWIx_MASTER_ADDR Register	29-44
TWIx_MASTER_STAT Register	29-45
TWIx_FIFO_CTL Register	29-45
TWIx_FIFO_STAT Register	29-47
TWIx_INT_MASK Register	29-47

Contents

TWI _x _INT_STAT Register	29-51
TWI _x _XMT_DATA8 Register	29-52
TWI _x _XMT_DATA16 Register	29-52
TWI _x _RCV_DATA8 Register	29-53
TWI _x _RCV_DATA16 Register	29-54
Programming Examples	29-55
Master Mode Setup	29-55
Slave Mode Setup	29-60
Electrical Specifications	29-67

SPORT CONTROLLERS

Overview	30-1
Interface Overview	30-3
SPORT Pin/Line Terminations	30-10
Description of Operation	30-11
SPORT Operation	30-11
SPORT Disable	30-11
Setting SPORT Modes	30-12
Stereo Serial Operation	30-13
Multichannel Operation	30-17
Multichannel Enable	30-19
Frame Syncs in Multichannel Mode	30-20
Multichannel Frame	30-22
Multichannel Frame Delay	30-23
Window Size	30-23

Window Offset	30-24
Other Multichannel Fields in SPORT _x _MCMC2	30-24
Channel Selection Register	30-24
Multichannel DMA Data Packing	30-26
Support for H.100 Standard Protocol	30-27
2X Clock Recovery Control	30-27
Functional Description	30-28
Clock and Frame Sync Frequencies	30-28
Maximum Clock Rate Restrictions	30-29
Word Length	30-29
Bit Order	30-30
Data Type	30-30
Companding	30-31
Clock Signal Options	30-31
Frame Sync Options	30-32
Framed Versus Unframed	30-32
Internal Versus External Frame Syncs	30-34
Active Low Versus Active High Frame Syncs	30-35
Sampling Edge for Data and Frame Syncs	30-35
Early Versus Late Frame Syncs (Normal Versus Alternate Timing)	30-37
Data Independent Transmit Frame Sync	30-39
Moving Data Between SPORTs and Memory	30-40
SPORT RX, TX, and Error Interrupts	30-40
PAB Errors	30-41

Contents

Timing Examples	30-41
SPORT Registers	30-48
Register Writes and Effective Latency	30-50
SPORT _x _TCR1 and SPORT _x _TCR2 Registers	30-51
SPORT _x _RCR1 and SPORT _x _RCR2 Registers	30-56
Data Word Formats	30-61
SPORT _x _TX Register	30-61
SPORT _x _RX Register	30-63
SPORT _x _STAT Register	30-66
SPORT _x _TCLKDIV and SPORT _x _RCLKDIV Registers	30-68
SPORT _x _TFSDIV and SPORT _x _RFSDIV Register	30-69
SPORT _x _MCMC _n Registers	30-70
SPORT _x _CHNL Register	30-71
SPORT _x _MRCS _n Registers	30-72
SPORT _x _MTCS _n Registers	30-74
Programming Examples	30-76
SPORT Initialization Sequence	30-77
DMA Initialization Sequence	30-78
Interrupt Servicing	30-81
Starting a Transfer	30-82

UART PORT CONTROLLERS

Overview	31-1
Features	31-2
Interface Overview	31-3

External Interface	31-4
Internal Interface	31-5
Description of Operation	31-6
UART Transfer Protocol	31-6
UART Transmit Operation	31-7
UART Receive Operation	31-9
Hardware Flow Control	31-12
IrDA Transmit Operation	31-14
IrDA Receive Operation	31-15
Interrupt Processing	31-17
Bit Rate Generation	31-19
Autobaud Detection	31-21
Programming Model	31-23
Non-DMA Mode	31-23
DMA Mode	31-25
Mixing Modes	31-27
UART Registers	31-28
UART _x _LCR Registers	31-30
UART _x _MCR Registers	31-33
UART _x _LSR Registers	31-36
UART _x _MSR Registers	31-39
UART _x _THR Registers	31-41
UART _x _RBR Registers	31-42
UART _x _IER_SET and UART _x _IER_CLEAR Registers	31-43

Contents

UARTx_DLL and UARTx_DLH Registers	31-48
UARTx_SCR Registers	31-49
UARTx_GCTL Registers	31-50
Programming Examples	31-51

USB OTG CONTROLLER

Overview	32-2
Features	32-2
Interface Overview	32-3
FIFO Configuration	32-7
Interrupts	32-8
Resets	32-11
Description of Operation	32-12
Peripheral Mode Operation	32-13
Endpoint Setup	32-13
IN Transactions as a Peripheral	32-14
High Bandwidth Isochronous IN Endpoints	32-16
OUT Transactions as a Peripheral	32-17
High Bandwidth Isochronous OUT Endpoints	32-19
Peripheral Transfer Workflows	32-20
Control Transactions as a Peripheral	32-21
Write Requests	32-22
Zero Data Requests	32-23
Peripheral Mode, Bulk IN, Transfer Size Known	32-24
Peripheral Mode, Bulk IN, Transfer Size Unknown	32-25

Peripheral Mode, ISO IN, Small MaxPktSize	32-26
Peripheral Mode, ISO IN, Large MaxPktSize	32-26
Peripheral Mode, Bulk OUT, Transfer Size Known	32-27
Peripheral Mode, Bulk OUT, Transfer Size Unknown ...	32-28
Peripheral Mode, ISO OUT, Small MaxPktSize	32-29
Peripheral Mode, ISO OUT, Large MaxPktSize	32-29
Peripheral Mode Suspend	32-30
Start Of Frame (SOF) Packets	32-30
Soft Connect / Soft Disconnect	32-31
Error Handling As a Peripheral	32-31
STALLS Issued to Control Transfers	32-33
Zero Length OUT Data Packets in Control Transfers	32-33
Host Mode Operation	32-34
Endpoint Setup and Data Transfer	32-34
Control Transaction as a Host	32-34
Setup Phase as a Host	32-35
IN Data Phase as a Host	32-36
OUT Data as a Host (Control)	32-37
IN Status Phase, (SETUP phase or OUT Data Phase)	32-39
OUT Status Phase as a Host (following IN Data Phase) ...	32-39
Host IN Transactions	32-40
Host OUT Transactions	32-41
Transaction Scheduling	32-42
Babble	32-43

Contents

Host Mode Reset	32-43
Host Mode Suspend	32-43
Functional Description	32-44
On-Chip Bus Interfaces	32-44
Interface Pins	32-45
Power and Clocking	32-45
UTMI Interface	32-46
Programming Model	32-46
OTG Session Request	32-47
Starting a Session	32-47
Detecting Activity	32-48
Host Negotiation/Configuration	32-49
Software Clock Control	32-50
Wakeup from Hibernate State	32-50
Wakeup without Re-Enumeration	32-53
Data Transfer	32-55
Loading/Unloading Packets from Endpoints	32-56
DMA Master Channels	32-57
DMA Bus Cycles	32-59
Transferring Packets Using DMA	32-59
Individual Packet: Rx Endpoint	32-60
Individual Packet: TX Endpoint	32-61
Multiple Packets: Rx Endpoint	32-61
Multiple Packets: TX Endpoints	32-63

USB OTG Registers	32-64
USB Function Address (USB_FADDR) Register	32-81
USB Power Management (USB_POWER) Register	32-82
USB Transmit Interrupt (USB_INTRTX) Register	32-85
USB Receive Interrupt (USB_INTRRX) Register	32-86
USB Transmit Interrupt Enable (USB_INTRTXE) Register .	32-87
USB Receive Interrupt Enable (USB_INTRRXE) Register	32-88
USB Common Interrupts (USB_INTRUSB) Register	32-89
USB Common Interrupt Enable (USB_INTRUSBE) Register	32-90
USB Frame Number (USB_FRAME) Register	32-91
USB Index (USB_INDEX) Register	32-91
USB Test Mode (USB_TESTMODE) Register	32-93
USB Global Interrupt (USB_GLOBINTR) Register	32-94
USB Global Control (USB_GLOBAL_CTL) Register	32-95
USB Tx Max Packet (USB_TX_MAX_PACKET) Register ...	32-97
USB Control/Status EP0 (USB_CSR0) Register	32-98
USB Tx Control/Status EPx (USB_TXCSR) Register	32-102
USB Rx Max Packet (USB_RX_MAX_PACKET) Register .	32-107
USB Rx Control/Status (USB_RXCSR) Register	32-109
USB Count 0 (USB_COUNT0) Register	32-115
USB Rx Byte Count EPx (USB_RXCOUNT) Register	32-116
USB Tx Type (USB_TXTYPE) Register	32-117
USB NAK Limit 0 (USB_NAKLIMIT0) Register	32-117
USB Tx Interval (USB_TXINTERVAL) Register	32-118

Contents

USB Rx Type (USB_RXTYPE) Register	32-119
USB Rx Interval (USB_RXINTERVAL) Register	32-120
USB Tx Byte Count EPx (USB_TXCOUNT) Register	32-121
USB Endpoint FIFO (USB_EPx_FIFO) Registers	32-122
USB OTG Device Control Register	32-122
USB OTG VBUS Interrupt Register	32-124
USB OTG VBUS Mask Register	32-126
USB Link Info (USB_LINKINFO) Register	32-127
USB VBUS Pulse Length (USB_VPLEN) Register	32-127
USB High-Speed EOF 1 (USB_HS_EOF1) Register	32-128
USB Full-Speed EOF 1 (USB_FS_EOF1) Register	32-128
USB Low-Speed EOF 1 (USB_LS_EOF1) Register	32-129
USB APHY Control 2 (USB_APHY_CNTRL2) Register ...	32-130
USB PLL OSC Control (USB_PLLOSC_CTRL) Registers	32-132
USB SRP Clock Divider (USB_SRP_CLKDIV) Register ...	32-133
USB DMA Interrupt (USB_DMA_INTERRUPT) Register	32-134
USB DMAx Control (USB_DMA_CONTROL) Registers .	32-135
USB DMAx Address Low Registers	32-137
USB DMAx Address HighRegisters	32-138
USB DMAx Count Low Registers	32-139
USB DMAx Count High Registers	32-140
Programming Examples	32-141
References	32-141
Glossary of USB Terms	32-141

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers	A-3
System Reset and Interrupt Control Registers	A-3
Watchdog Timer Registers	A-3
Real-Time Clock Registers	A-4
Timer Registers	A-4
Ports Registers	A-4
External Bus Interface Unit Registers	A-4
DMA/Memory DMA Control Registers	A-5
Handshake MDMA Control Registers	A-5
Host DMA Registers	A-5
PIXC Registers	A-5
Rotary Counter Registers	A-5
Security Registers	A-6
Core Timer Registers	A-6
Processor-Specific Memory Registers	A-6
MXVR Registers	A-7
Keypad Registers	A-13
SDH Registers	A-13
ATAPI Registers	A-16
NAND Flash Controller Registers	A-18
EPPI1 Registers	A-19
EPPI2 Registers	A-20

Contents

CANx Registers	A-22
SPI0 Controller Registers	A-32
SPI1 Controller Registers	A-32
TWI Registers	A-33
SPORT0 Controller Registers	A-35
SPORT1 Controller Registers	A-37
SPORT2 Controller Registers	A-39
SPORT3 Controller Registers	A-41
UART0 Controller Registers	A-43
UART1 Controller Registers	A-44
UART2 Controller Registers	A-45
UART3 Controller Registers	A-46
USB OTG Registers	A-47

TEST FEATURES

JTAG Standard	B-1
Boundary-Scan Architecture	B-3
Instruction Register	B-5
Public Instructions	B-5
EXTEST – Binary Code 00000	B-5
SAMPLE/PRELOAD – Binary Code 10000	B-7
BYPASS – Binary Code 11111	B-7
IDCODE – Binary Code 00010	B-7
Boundary-Scan Register	B-7

PREFACE

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

Contents of Two Volumes

Contents of Volume 1 and Volume 2 are listed below.

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

Purpose of This Manual

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

Intended Audience

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

Manual Contents

This manual consists of:

- Chapter 20, “[Introduction](#)”
Provides a high level overview of the processor, including peripherals, power management, and development tools.
- Chapter 21, “[Media Transceiver Module \(MXVR\)](#)”
Describes the media transceiver module (MXVR) of the processor. This transceiver serves as the network interface to a media-oriented system transport (MOST[®]) ring network.
- Chapter 22, “[Keypad Interface](#)”
Describes the keypad interface of the processor. This interface is a 16-pin interface module which is used to detect the key pressed in a 8x8 (maximum) keypad matrix. The size of the input keypad matrix is programmable.
- Chapter 23, “[Secure Digital Host](#)”
Describes the secure digital input/output cards (SDIO) interface of the processor. The SDIO support includes support for the interrupt and read/wait signals for SDIO.
- Chapter 24, “[ATAPI Interface](#)”
Describes the ATAPI interface of the processor. This interface is an ATA/ATAPI-6 compliant host implementation. The ATA (AT attachment) interface, also known as IDE (integrated drive electronics) interface, provides a simple interface to low cost, non-volatile memories like hard-disk drives, DVD players, CD-ROM players/writers, and Compact Flash and PC-Card devices.

Manual Contents

- Chapter 25, “[NAND Flash Controller](#)”
Describes the NAND flash controllers (NFC)—which are part of the external bus interface—of the processor. NAND flash devices provide high density, low cost memory.
- Chapter 26, “[Enhanced Parallel Peripheral Interface](#)”
Describes the enhanced parallel peripheral interfaces (EPPIx) of the processor. The EPPI is a half-duplex, bidirectional port accommodating up to 24 bits of data and is used for digital video and data converter applications.
- Chapter 27, “[CAN Module](#)”
Describes the Controller Area Network (CANx) modules, which are low bit rate serial interfaces intended for use in applications where bit rates are typically up to 1M bit/s.
- Chapter 28, “[SPI-Compatible Port Controllers](#)”
Describes the serial peripheral interface (SPIx) ports that provide an I/O interface to a variety of SPI-compatible peripheral devices.
- Chapter 29, “[Two Wire Interface Controllers](#)”
Describes the two-wire interface (TWIx) controllers, which allows a device to interface to an Inter IC bus as specified by the *Philips I²C Bus Specification version 2.1* dated January 2000.
- Chapter 30, “[SPORT Controllers](#)”
Describes the independent, synchronous serial port controllers (SPORT0, SPORT1, SPORT2, and SPORT3) that provide an I/O interface to a variety of serial peripheral devices.
- Chapter 31, “[UART Port Controllers](#)”
Describes the universal asynchronous receiver/transmitter ports (UART0, UART1, UART2 and UART3) that convert data between serial and parallel formats. The UARTs support the half-duplex, IrDA® SIR protocol as a mode-enabled feature.

- Chapter 32, “[USB OTG Controller](#)”
Describes the USB OTG interface of the processor. This interface 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 through a point-to-point USB connection without the need for a PC host.
- Appendix A, “[System MMR Assignments](#)”
Lists the memory-mapped registers included in this manual, their addresses, and cross-references to text.
- Appendix B, “[Test Features](#)”
Describes test features for the processor, discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.

What’s New in This Manual

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

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at <http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to processor.tools.support@analog.com

Technical or Customer Support

- E-mail processor questions to
processor.support@analog.com (World wide support)
processor.europe@analog.com (Europe support)
processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

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

Blackfin (ADSP-BFxxx) Processors

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

TigerSHARC® (ADSP-TSxxx) Processors




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


SHARC® (ADSP-21xxx) Processors

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

Conventions


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

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

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

Register Diagram Conventions

Register diagrams use the following conventions:

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

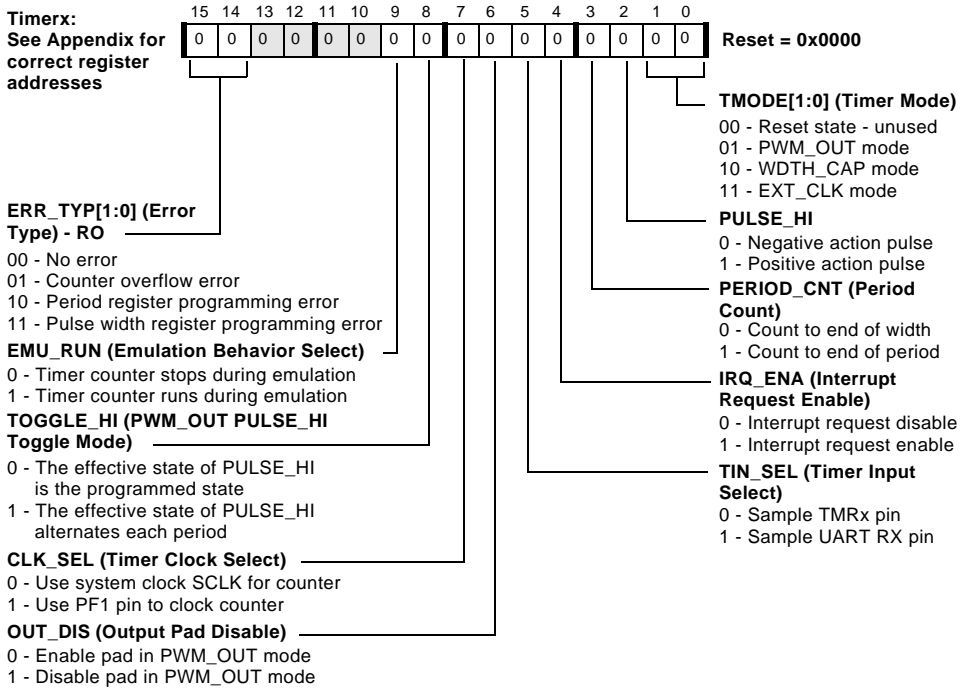
Examples of these conventions are shown in [Figure P-1](#).

Conventions

Table P-1. Short Form of Register Names

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

Timer Configuration Registers (TIMERx_CONFIG)



Core Timer Count Register (TCOUNT)

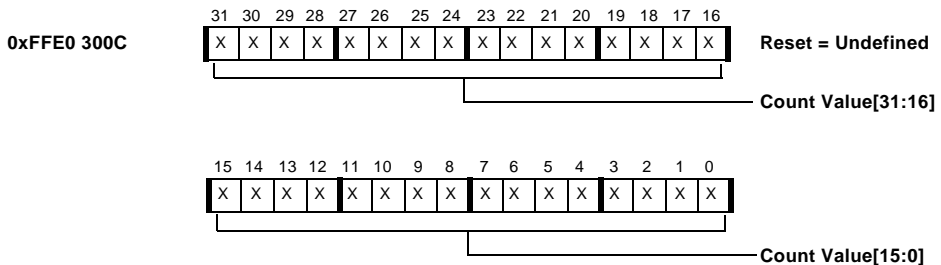


Figure P-1. Register Diagram Examples

Conventions

20 INTRODUCTION

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

This chapter includes the following sections:

- [“Peripherals” on page 20-3](#)
- [“Memory Architecture” on page 20-6](#)
- [“DMA Support” on page 20-11](#)
- [“External Bus Interface Unit” on page 20-14](#)
- [“Ports” on page 20-15](#)
- [“Two-Wire Interfaces” on page 20-16](#)
- [“Controller Area Network” on page 20-17](#)
- [“Enhanced Parallel Peripheral Interface \(EPPI\)” on page 20-18](#)
- [“SPORT Controllers” on page 20-20](#)
- [“Serial Peripheral Interface \(SPI\) Ports” on page 20-22](#)
- [“Timers” on page 20-22](#)

- “UART Ports” on page 20-23
- “USB On-The-Go, Dual-Role Device Controller” on page 20-24
- “ATA/ATAPI-6 Interface” on page 20-25
- “Keypad Interface” on page 20-25
- “Secure Digital (SD)/SDIO Controller” on page 20-26
- “Rotary Counter and Thumbwheel Interface” on page 20-27
- “Security” on page 20-27
- “Media Transceiver (MXVR) MAC Layer” on page 20-29
- “Real-Time Clock” on page 20-30
- “Watchdog Timer” on page 20-31
- “Clock Signals” on page 20-32
- “Dynamic Power Management” on page 20-32
- “Voltage Regulation” on page 20-34
- “Boot Modes” on page 20-35
- “Instruction Set Description” on page 20-35
- “Development Tools” on page 20-36

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
- Up/down counter with support for rotary encoder
- Up to 152 general-purpose I/O (GPIOs)
- On-chip PLL capable of 0.5x to 64x frequency multiplication
- Debug/JTAG interface

Peripherals

These peripherals are connected to the core through several high bandwidth buses, as shown in [Figure 20-1](#).

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

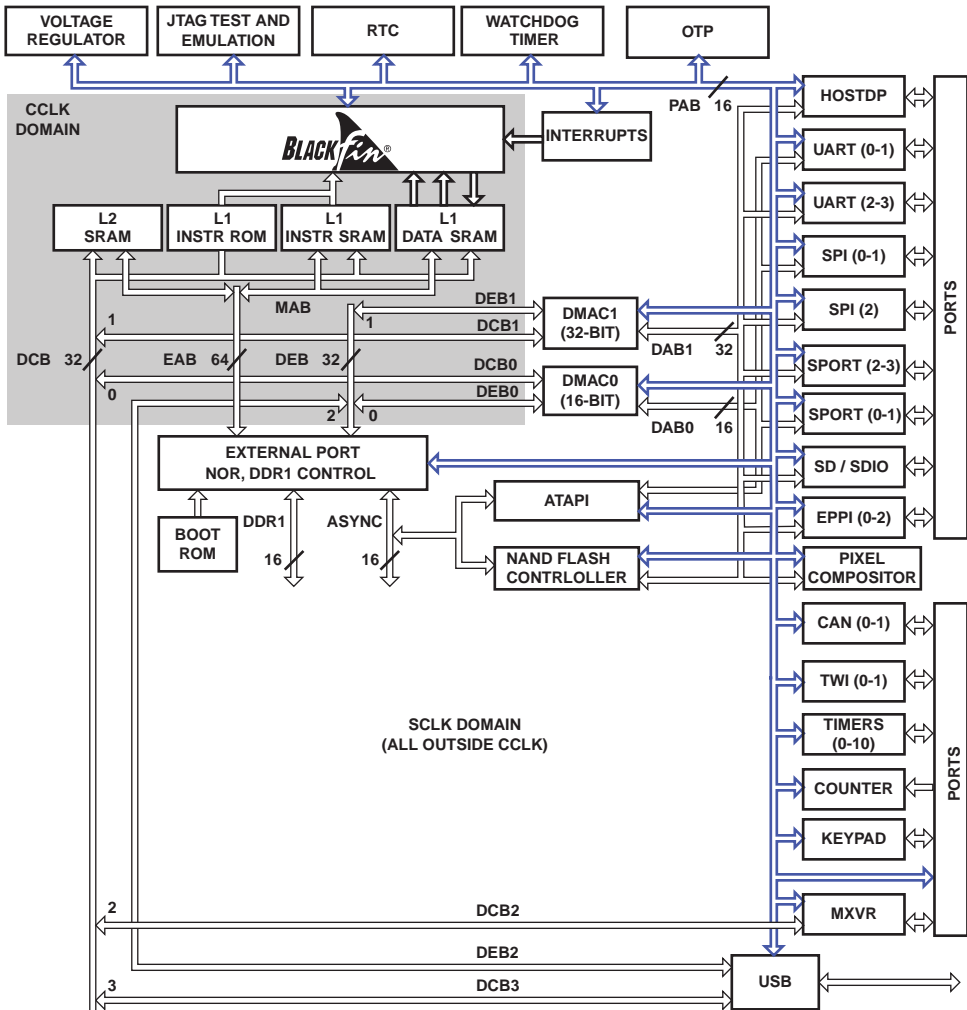


Figure 20-1. ADSP-BF54x Processor Block Diagram

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency, on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems. [Table 20-1](#) shows the memory comparison for the ADSP-BF54x processors.

Table 20-1. 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 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 L2 memory is a unified instruction and data memory and can hold any mixture of code and data required by the system design.

Internal Memory

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

- L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.
- L1 instruction ROM, operating at full processor speed. This ROM is not customer configurable.
- L2 SRAM, providing up to 128K bytes of unified instruction and data memory, operating at one half the frequency of the core.
- 4K boot ROM that can be seen as L3 memory. It operates at full SCLK rate.

External Memory

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

The DDR memory controller can gluelessly manage up to two banks of double-rate synchronous dynamic memory (DDR1 SDRAM). The 16-bit wide interface operates at `SCLK` frequency enabling maximum throughput of 532M byte/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 64M bit to 512M bit device sizes and 4-, 8-, or 16-bit widths. The controller supports up to 512M bytes in one bank, but the total in two banks is also limited to 512M bytes. 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 64M byte asynchronous memory banks, represented by four memory select strobes. Alternatively, these strobes can function as bank-specific read or write strobes preventing further glue logic when connecting to asynchronous FIFO devices.

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

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

NAND Flash Controller (NFC)

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

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

I/O Memory Space

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

One-Time-Programmable (OTP) Memory

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

The OTP is not part of the Blackfin linear memory map. OTP memory is not accessed directly using the Blackfin memory map, rather, it is accessed through four 32-bit 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 dependent 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 times (public OTP memory). The second area maintains privileged access and can only be accessed (read/write) upon entry into secure mode when security features are utilized (private OTP memory).

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

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

DMA Support

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

While the USB controller and MXVR have their own dedicated DMA controllers, the other on-chip peripherals are managed by two centralized DMA controllers, called DMAC1 (32-bit) and DMAC0 (16-bit). Both operate in the `SCLK` domain. Each DMA controller manages twelve independent DMA channels. The DMAC1 controller masters high-bandwidth peripherals over a dedicated 32-bit DMA access bus (DAB32). Similarly, the DMAC0 controller masters most of the 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 conflict, if the L1 memory is accessed by the core simultaneously. Similarly, there are dedicated DMA buses between the

DMA Support

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

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

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

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

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

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

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 handshaked memory DMA (HMDMA) scheme can be used to efficiently exchange data with block-buffered or FIFO-style devices connected externally. Users can select whether the DMA request pins control the source or the destination side of the memory DMA. It allows control of the number of data transfers for memory DMA. The number of transfers per edge is programmable. This feature can be programmed to allow memory DMA to have an increased priority on the external bus relative to the core.

Host DMA Interface

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

The HOSTDP port is enabled through the peripheral access bus. Once enabled, the DMA is controlled by the external host. The external host can then program the DMA to send/receive data to any valid internal and 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

External Bus Interface Unit

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

External Bus Interface Unit

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

DDR SDRAM Controller

The DDR memory controller can gluelessly manage up to two banks of double-rate synchronous dynamic memory (DDR1 SDRAM). The 16-bit interface operates at *SCLK* frequency enabling a maximum throughput of 532 Mbyte/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 bits (64MByte). 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 = 265M byte per external bank can be supported. ADSP-BF54x Blackfin processor’s two external banks provide support for a maximum of 2 x 256M bytes = 512M bytes.

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 1M byte window in the processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks are 16 bits wide, for interfacing to a range of memories and I/O devices.

Ports

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

General-Purpose I/O (GPIO)

Every pin in Port A to Port J can function as a GPIO pin resulting in a GPIO pin count of up to 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 peripheral 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

Two-Wire Interfaces

write-one-to-set and write-one-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 Interfaces

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

The TWI externally moves 8-bit data while maintaining compliance with the I²C bus protocol. The *Philips I²C Bus Specification version 2.1* covers many variants of I²C. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multimaster data arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lockup

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

Controller Area Network

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

The ADSP-BF54x CAN controllers offer:

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

Enhanced Parallel Peripheral Interface (EPPI)

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

Enhanced Parallel Peripheral Interface (EPPI)

The ADSP-BF54x processor provides multiple enhanced parallel peripheral interfaces (EPPIs) - 18-/24-bit PPI0 with LCD, 16-bit PPI1, and 8-bit PPI2. The EPPI supports the direct connection to active TFT LCDs, 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, or 24 bits per clock cycle
- PPI0 can connect to 18-bit and 24-bit RGB LCD displays. PPI1 can support up to 16-bit data, or be split into two independent 8-bit EPPIs (PPI1/PPI2)
- PPI0 can be configured for data lengths of 8, 10, 12, 14, 16, 18 or 24 bits. PPI1 can be configured for data lengths of 8, 10, 12, 14, or 16 bits. PPI2 supports only 8-bit data
- Bidirectional and half-duplex port
- `PPIx_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 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

SPORT Controllers

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

- I²S capable operation

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

- Buffered (eight-deep) transmit and receive ports

Each port has a data register for transferring data words to and from other processor components and shift registers for shifting data in and out of the data registers.

- Clocking

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

- Word length

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

- 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) Ports

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

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

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

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

Timers

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

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

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

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

UART Ports

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

- Programmed I/O

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

- Direct Memory Access (DMA)

The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. Each of the two UARTs have two

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

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 `UARTx_RTS` (request to send) and `UARTx_CTS` (clear to send) signals for hardware flow purposes. The transmitter hardware is automatically prevented from sending further data when the `UARTx_CTS` input is deasserted. The receiver can automatically deassert its `UARTx_RTS` 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 On-The-Go (OTG) Dual-Role Device controller (USBDRD) provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data using a point-to-point USB connection without

the need for a PC host. The USBDR module can operate in a traditional USB peripheral-only mode as well as the host mode presented in the on-the-go (OTG) supplement to the USB 2.0 specification. In host mode, the USB module supports transfers at high-speed (480 Mbps), full-speed (12 Mbps), and low-speed (1.5 Mbps) rates. Peripheral-only mode supports the high and full speed transfer rates.

ATA/ATAPI-6 Interface

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

- Supports PIO modes 0, 1, 2, 3, and 4
- Supports multiword DMA modes 0, 1, 2 and
- Supports ultra DMA modes 0, 1, 2, 3, 4, and 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 8x8 (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

Secure Digital (SD)/SDIO Controller

width of the filtered bounce is programmable. The keypad 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, release, and press of a key as two consecutive presses of the same key, where the latter 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:

- Maximum of 8x8 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 10 M 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 (Cyclical Redundancy Checking).

Rotary Counter and Thumbwheel Interface

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

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

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

Security

The ADSP-BF54x Blackfin processor provides security features (Blackfin Lockbox™ Secure Technology) that enable customer applications to use secure protocols consisting of code authentication and execution of code within a secure environment. Implementing secure protocols on Blackfin

Security

processors involves 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 (MXVR) MAC Layer

The ADSP-BF54x Blackfin 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 transfers. It offers faster lock times, greater jitter immunity and a sophisticated DMA scheme for data transfers. The high speed internal interface to the core and L1 memory allows the full bandwidth of the network to be utilized. The MXVR can operate as either the network master or as a network slave.

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

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

Real-Time Clock

from sleep mode or the hibernate state. These features allow the ADSP-BF54x to operate in a low-power state when there is no network activity or when data is not currently being received or transmitted by the MXVR.

The MXVR clock is provided through a dedicated external crystal or crystal oscillator. The frequency of the external crystal or the crystal oscillator can be $256F_s$, $384F_s$, $512F_s$, or $1024F_s$ for $F_s = 38$ 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 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, and the RTC is the only peripheral capable of waking the processor from deep sleep mode.

Watchdog Timer

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

If configured to generate a hardware reset, the watchdog timer resets both the core and the ADSP-BF54x processor peripherals, but it does not reset the dynamic power management controller. 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).

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 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 register 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 L2 memory is not supported.

Voltage Regulation

Deep Sleep Mode (Maximum Dynamic Power Savings)

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

Hibernate State (Maximum Power Savings)

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

Voltage Regulation

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

The regulator can also be disabled and bypassed at the user's discretion. For more information, see the voltage regulator circuit diagram in the *ADSP-BF54x Blackfin Embedded Processor* data sheet.

Boot Modes

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 as well as special commands that instruct the boot kernel on 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 to jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called booting.

Instruction Set Description

The ADSP-BF54x processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to the *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.

Development Tools

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 plus two load/store plus two pointer updates per cycle
- All registers, I/O, and memory-mapped into a unified 4G byte memory space, providing a simplified programming model

Code density enhancements include intermixing of 16- and 32-bit instructions with no mode switching or code segregation. Frequently used instructions are encoded in 16 bits.

Development Tools

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

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

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

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

Development Tools

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

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

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

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

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks.

Nonintrusive in-circuit emulation is assured by the use of the processor's JTAG interface—the emulator does not affect target system loading or timing.

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

21 MEDIA TRANSCEIVER MODULE (MXVR)

Among the ADSP-BF54x Blackfin processors, the MXVR is only available on the ADSP-BF549 processor.

This chapter includes the following sections:

- [“Overview” on page 21-1](#)
- [“Interface Signals” on page 21-3](#)
- [“MXVR Memory Map” on page 21-5](#)
- [“MXVR Registers” on page 21-6](#)
- [“General Operation” on page 21-115](#)

Overview

The media transceiver module (MXVR) serves as the network interface to a media-oriented system transport (MOST[®]) ring network for the ADSP-BF54x 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 net interface.

Overview

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 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 core. Synchronous data can be transferred to and from synchronous channels through eight 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 nodes or other MOST transceivers.

Interface Signals

Table 21-1 on page 21-4 lists the MXVR signal pins. All output pins are 3.3V compliant. The $\overline{\text{MRX}}$ and $\overline{\text{MRXON}}$ 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 GPIO module. For more information, see the “General Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Interface Signals

Table 21-1. 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 21-2](#) 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 21-2. 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-1 on page A-7](#) 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 21-3 lists the MXVR registers.

Table 21-3. MXVR Registers

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2700	MXVR_CONFIG	MXVR configuration register on page 21-12	16 R/W	0x1FCA
0xFFC0 2704	Reserved	–	–	–
0xFFC0 2708 0xFFC0 270C	MXVR_STATE_0 MXVR_STATE_1	MXVR state registers on page 21-19	32 RO	0x0000 0000
0xFFC0 2710 0xFFC0 2714	MXVR_INT_STAT_0 MXVR_INT_STAT_1	MXVR interrupt registers on page 21-29	32 R/W	0x0000 0000
0xFFC0 2718 0xFFC0 271C	MXVR_INT_EN_0 MXVR_INT_EN_1	MXVR interrupt enable registers on page 21-43	32 R/W	0x0000 0000
0xFFC0 2720	MXVR_POSITION	MXVR node position register on page 21-48	16 RO	0x8000
0xFFC0 2724	MXVR_MAX_POSITION	MXVR maximum node position register on page 21-50	16 RO	0x0000
0xFFC0 2728	MXVR_DELAY	MXVR node frame delay register on page 21-51	16 RO	0x8000
0xFFC0 272C	MXVR_MAX_DELAY	MXVR maximum node frame delay register on page 21-53	16 RO	0x0000
0xFFC0 2730	MXVR_LADDR	MXVR logical address register on page 21-54	32 R/W	0x0000 0FFF
0xFFC0 2734	MXVR_GADDR	MXVR group address register on page 21-55	16 R/W	0x0000
0xFFC0 2738	MXVR_AADDR	MXVR alternate address register on page 21-56	32 R/W	0x0000 0FFF

Media Transceiver Module (MXVR)

Table 21-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
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 registers on page 21-56	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 data logical channel assignment registers on page 21-58	32 R/W	0xFFFF FFFF
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 DMA configuration registers on page 21-60	32 R/W	0x0000 0000

MXVR Registers

Table 21-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
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 start address registers on page 21-70	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 transfer count registers on page 21-73	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 current address registers on page 21-72	32 RO	0xFF00 0000
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 current transfer count registers on page 21-76	16 RO	0x0000
0xFFC0 2838	MXVR_AP_CTL	MXVR asynchronous packet control register on page 21-77	16 R/W	0x0000

Media Transceiver Module (MXVR)

Table 21-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 283C	MXVR_APRB_START_ADDR	MXVR asynchronous packet receive buffer start address register on page 21-82	32 R/W	0xFF00 0000
0xFFC0 2840	MXVR_APRB_CURR_ADDR	MXVR asynchronous packet receive buffer current address register on page 21-81	32 RO	0xFF00 0000
0xFFC0 2844	MXVR_APTB_START_ADDR	MXVR asynchronous packet transmit buffer start address register on page 21-82	32 R/W	0xFF00 0000
0xFFC0 2848	MXVR_APTB_CURR_ADDR	MXVR asynchronous packet transmit buffer current address register on page 21-83	32 RO	0xFF00 0000
0xFFC0 284C	MXVR_CM_CTL	MXVR control message control register on page 21-83	32 R/W	0x0000 0000
0xFFC0 2850	MXVR_CMRB_START_ADDR	MXVR control message receive buffer start address register on page 21-86	32 R/W	0xFF00 0000
0xFFC0 2854	MXVR_CMRB_CURR_ADDR	MXVR control message receive buffer current address register on page 21-87	32 RO	0xFF00 0000
0xFFC0 2858	MXVR_CMTB_START_ADDR	MXVR control message transmit buffer start address register on page 21-88	32 R/W	0xFF00 0000
0xFFC0 285C	MXVR_CMTB_CURR_ADDR	MXVR control message transmit buffer current address register on page 21-89	32 RO	0xFF00 0000

MXVR Registers

Table 21-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 2860	MXVR_RRDB_START_ADDR	MXVR remote read buffer start address register on page 21-90	32 R/W	0xFF00 0000
0xFFC0 2864	MXVR_RRDB_CURR_ADDR	MXVR remote read buffer current address register on page 21-91	32 RO	0xFF00 0000
0xFFC0 2868 0xFFC0 2870	MXVR_PAT_DATA_0 MXVR_PAT_DATA_1	MXVR pattern data registers on page 21-92	32 R/W	0x0000 0000
0xFFC0 286C 0xFFC0 2874	MXVR_PAT_EN_0 MXVR_PAT_EN_1	MXVR pattern enable registers on page 21-93	32 R/W	0x0000 0000
0xFFC0 2878 0xFFC0 287C	MXVR_FRAME_CNT_0 MXVR_FRAME_CNT_1	MXVR frame counter registers on page 21-94	16 R/W	0x0000
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 registers on page 21-95	32 WO	0xFFFF XXXX
0xFFC0 28BC	Reserved	—	—	—
0xFFC0 28C0	MXVR_BLOCK_CNT	MXVR block counter register on page 21-98	16 R/W	0x0000
0xFFC0 28C4 to 0xFFC0 28CC	Reserved	—	—	—
0xFFC0 28D0	MXVR_CLK_CTL	MXVR clock control register on page 21-100	32 R/W	0x0202 0003

Media Transceiver Module (MXVR)

Table 21-3. MXVR Registers (Cont'd)

Register Address	Register Name	Register Description	Size (Bits)	Reset Value
0xFFC0 28D4	MXVR_CDRPLL_CTL	MXVR clock/data recovery PLL control register on page 21-107	32 R/W	0x0502 0820
0xFFC0 28D8	MXVR_FMPLL_CTL	MXVR frequency multiply PLL control register on page 21-110	32 R/W	0x1900 1020
0xFFC0 28DC	MXVR_PIN_CTL	MXVR pin control register on page 21-112	16 R/W	0x0000
0xFFC0 28E0	MXVR_SCLK_CNT	MXVR system clock counter register on page 21-113	16 R/W	0x0000
0xFFC0 28E4 to 0xFFC0 28FF	Reserved	–	–	–

MXVR Registers

MXVR Configuration Register (MXVR_CONFIG)

The MXVR_CONFIG register sets the configuration of the MXVR node.

MXVR Configuration Register (MXVR_CONFIG)

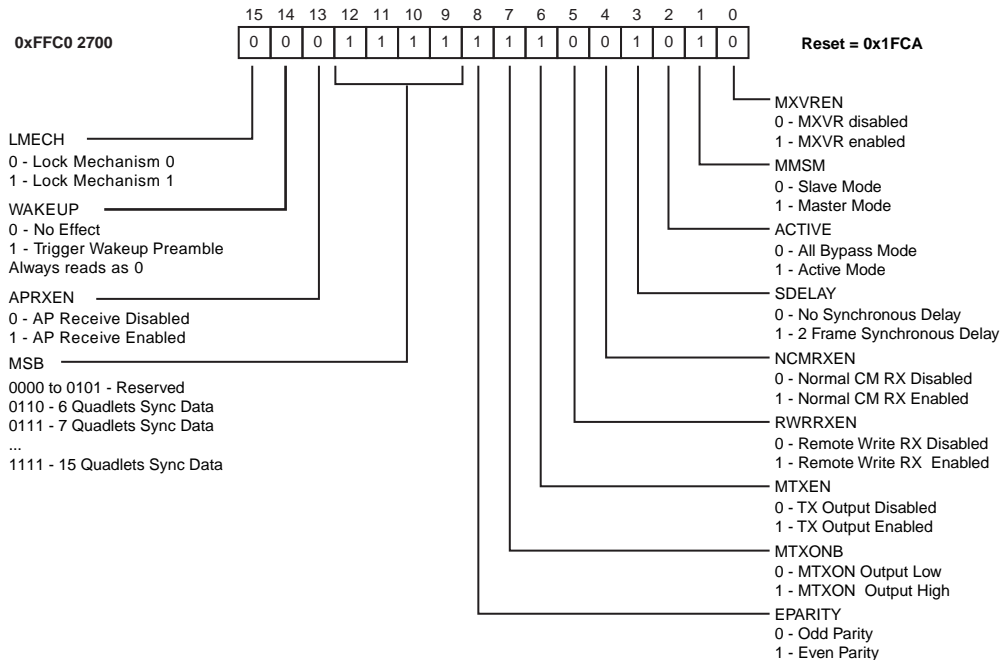


Figure 21-1. MXVR Configuration Register (MXVR_CONFIG)

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 $MDMAEN_x = 0$) and software should wait until all synchronous data DMA channels are inactive ($DMAACTIVE_x == 0$ and $DMAPMEN_x == 0$) 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 ($MXVREN = 1$

MXVR Registers

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 21-4 on page 21-14 lists all possible operating modes of the MXVR, the bit encodings to select the modes, and the functionality in the modes.

Table 21-4. 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
SD = Synchronous Data, AP = Asynchronous Packets, CM = Control Messages					

Table 21-4. MXVR Operating Modes (Cont'd)

Mode	MXVREN	MMSM	ACTIVE	SDELAY	Functionality
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.

MXVR Registers

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 asynchro-

nous 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.

MXVR Registers

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 Registers (MXVR_STATE_0, MXVR_STATE_1)

The MXVR_STATE_x registers indicate the current state of the MXVR. All bits in the MXVR_STATE_x registers are read-only bits.

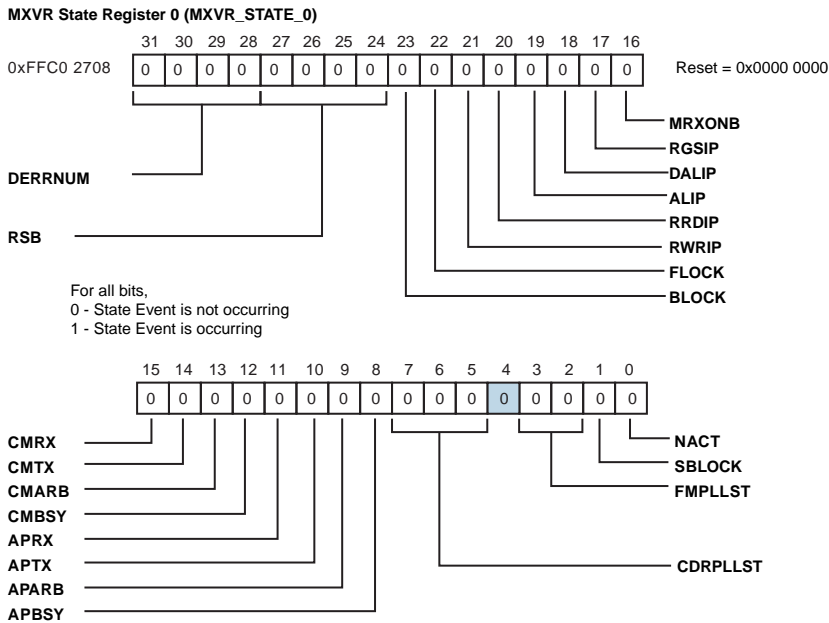


Figure 21-2. MXVR State Register (MXVR_STATE_0)

MXVR Registers

MXVR State Register 1 (MXVR_STATE_1)

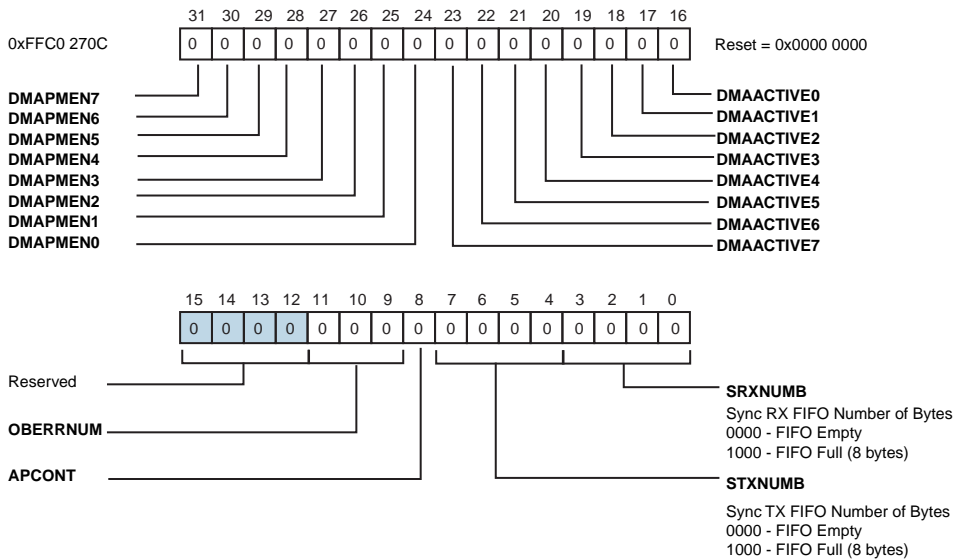


Figure 21-3. MXVR State Register (MXVR_STATE_1)

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 21-5](#).

Table 21-5. 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 21-6](#).

Table 21-6. 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

MXVR Registers

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.

MXVR Registers

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 from hibernate state.

The Remote Get Source In Progress (RGSIP) bit is a read-only bit which indicates whether a Remote Get Source 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.

MXVR Registers

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 21-7 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 21-7. 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 21-8](#) 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 21-8. 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

MXVR Registers

Table 21-8. DMA Out of Bounds Error Number Encodings

OBERRNUM	DMA Channel Causing Error
b#110	Synchronous Data DMA Channel 6
b#111	Synchronous Data DMA Channel 7

The DMAACTIVE_x bits indicate whether the DMA channel is active or inactive. When the DMAACTIVE_x bit is 1, DMA channel x is active and when the DMAACTIVE_x bit is 0, DMA channel x is inactive. Once the MDMAEN_x bit is set to 1, the exact time when the DMA goes active depends on the Flow Mode selected. When the MDMAEN_x 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 MDMAEN_x 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 MDMAEN_x 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 DMAPMEN_x 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 MDMAEN_x bit is set to 0, the DMA channel will go inactive and disable on the next frame boundary reached.

The DMAPMEN_x 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 MDMAEN_x 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.

MXVR Registers

MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)

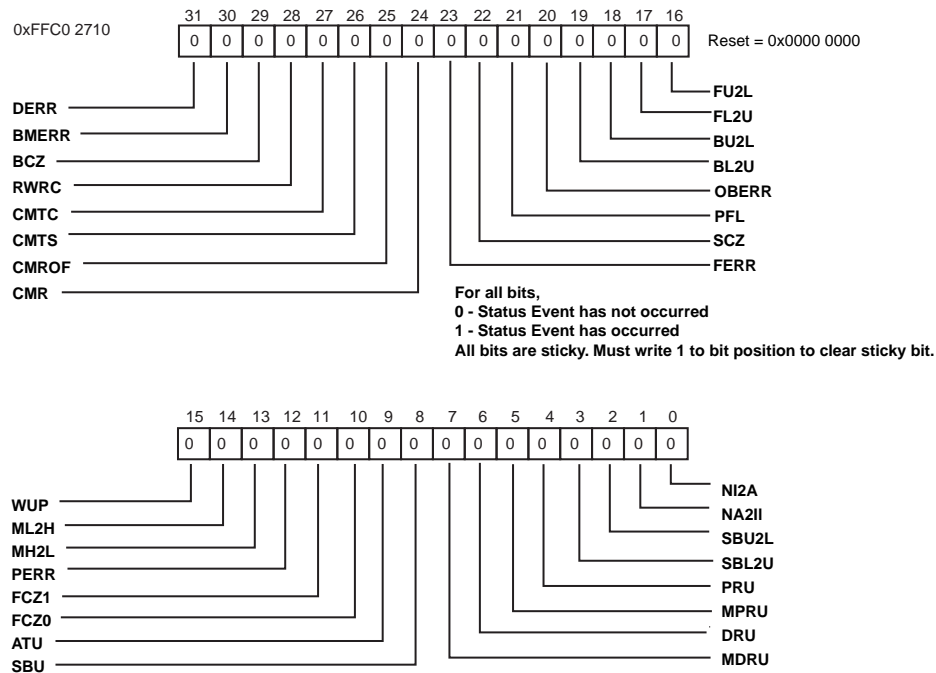


Figure 21-4. MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)

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 when 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

MXVR Registers

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 when 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 when 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 when 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 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.

MXVR Registers

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 FCZOEN 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=1). 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=1). 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

MXVR Registers


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 `OBERRREN` 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 `PFLLEN` 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 `FERR` 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 con-

MXVR Registers

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 `CMTCCEN` 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'd to the Remote Read Buffer and the received write address and write length have also been DMA'd 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.

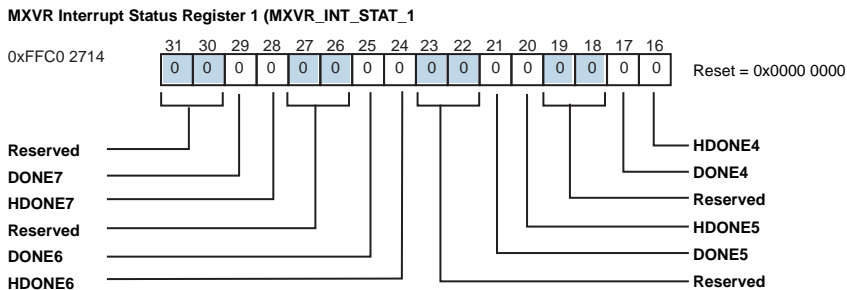
The Block Counter Zero (BCZ) interrupt event will change to 1 when the Block Counter is started by writing a value to the `MXVR_BLOCK_CNT` register and Block Counter has decremented down to zero. If the `BCZEN` bit is set to 1 in the `MXVR_INT_EN_0` register, the assertion of `BCZ` will generate a Status 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 Biphasic Mark Coding Error (BMERR) interrupt event will change to 1 when there is a biphasic 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 Register_1 (MXVR_INT_STAT_1)

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.



For all bits,
 0 - Status Event has not occurred
 1 - Status Event has occurred.
 All bits are sticky. Must write 1 to bit position to clear sticky bit.

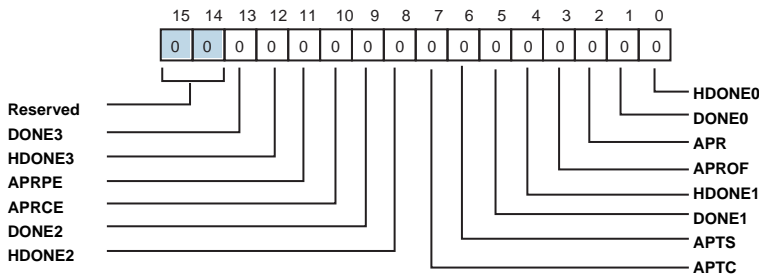


Figure 21-5. MXVR Interrupt Status Register_1 (MXVR_INT_STAT_1)

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 Registers

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 `APTSSEN` 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.

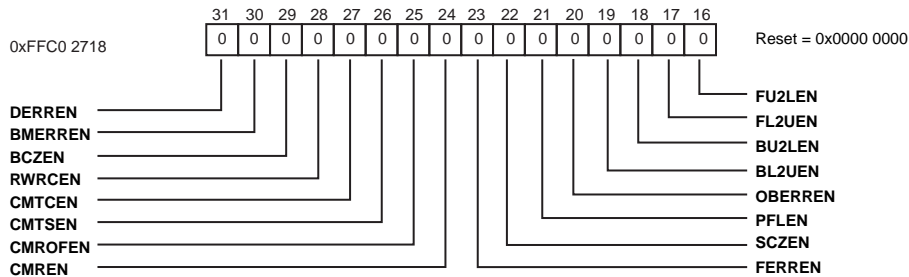
MXVR Interrupt Enable Register 0 (`MXVR_INT_EN_0`)

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

MXVR Registers

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.

MXVR Interrupt Enable Register 0 (MXVR_INT_EN_0)



For all bits,
 0 - Do not interrupt when Status Event occurs
 1 - Interrupt when Status Event occurs

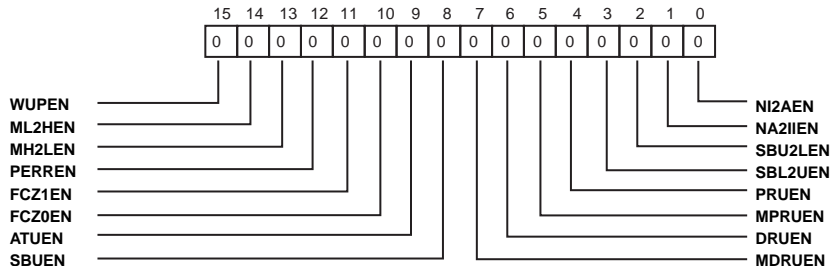


Figure 21-6. MXVR Interrupt Enable Register 0 (MXVR_INT_EN_0)

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.

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)

Media Transceiver Module (MXVR)

- 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 (PFLLEN)
- System Clock Counter Zero interrupt enable (SCZEN)
- FIFO Error interrupt enable (FERREN)
- Frame Counter 0 Zero interrupt enable (FCZOEN)

MXVR Registers

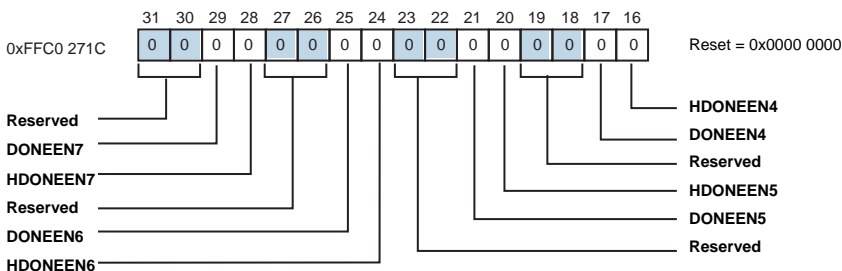
- Frame Counter 1 Zero interrupt enable (FCZ1EN)
- Control Message Received interrupt enable (CMREN)
- Control Message Receive Buffer Overflow interrupt enable
- Control Message Transmit Buffer Successfully Sent interrupt enable (CMTSEN)
- Control Message Transmit Buffer Successfully Cancelled interrupt 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 Register 1 (MXVR_INT_EN_1)

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.

MXVR Interrupt Enable Register 1 (MXVR_INT_EN_1)



For all bits,
 0 - Status Event has not occurred
 1 - Status Event has occurred.
 All bits are sticky. Must write 1 to bit position to clear sticky bit.

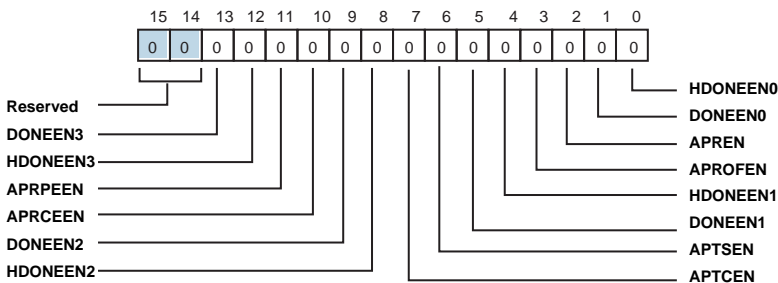


Figure 21-7. MXVR Interrupt Enable Register 1 (MXVR_INT_EN_1)

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.

MXVR Registers

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 Register (`MXVR_POSITION`)

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.

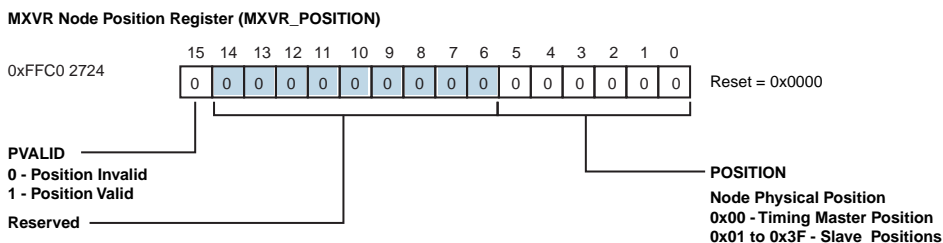


Figure 21-8. MXVR Node Position Register (MXVR_POSITION)

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 Register (MXVR_MAX_POSITION)

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).

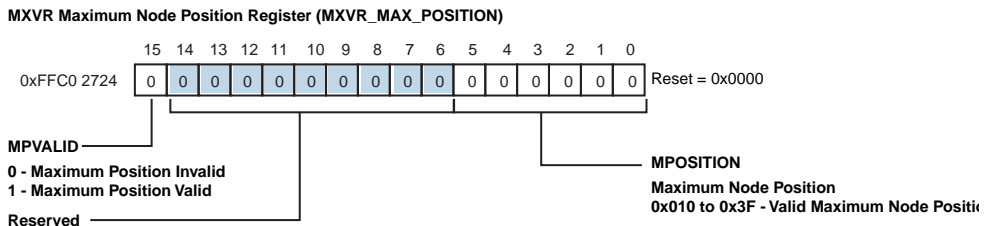


Figure 21-9. MXVR Maximum Node Position Register (MXVR_MAX_POSITION)

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 Register (MXVR_DELAY)

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 the MXVR. 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.

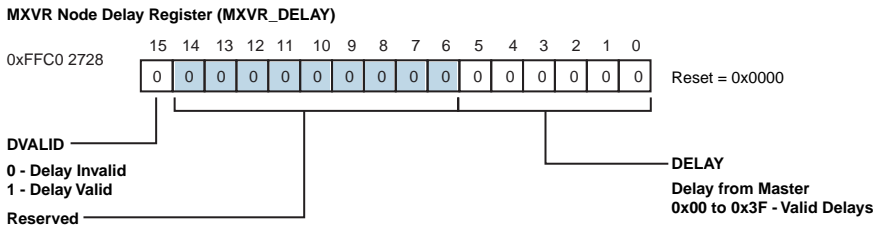


Figure 21-10. MXVR Node Frame Delay Register (MXVR_DELAY)

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

MXVR Registers

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`),

$$t_{\text{delay}} = 2 * (\text{DELAYA} - \text{DELAYB}) * (1 / F_s)$$

If (`POSITIONA > POSITIONB`) and (`SDELAYA == "0"`),

$$t_{\text{delay}} = 2 * (\text{MDELAY} - \text{DELAYA} + \text{DELAYB}) * (1 / F_s)$$

If (`POSITIONA > POSITIONB`) and (`SDELAYA == "1"`),

$$t_{\text{delay}} = 2 * (\text{MDELAY} - \text{DELAYA} + \text{DELAYB} - 1) * (1 / F_s)$$

MXVR Maximum Node Frame Delay Register (MXVR_MAX_DELAY)

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.

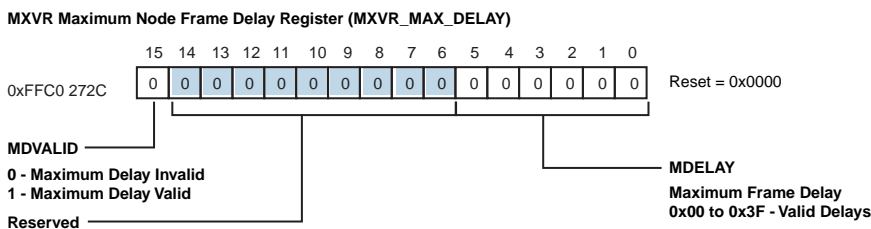


Figure 21-11. MXVR Maximum Node Frame Delay Register (MXVR_MAX_DELAY)

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

MXVR Registers

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 Register (MXVR_LADDR)

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.

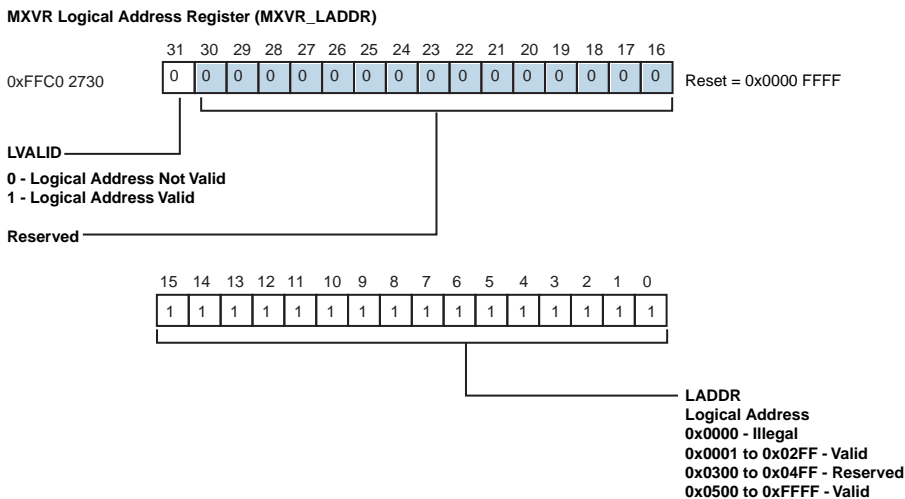


Figure 21-12. MXVR Logical Address Register (MXVR_LADDR)

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 Register (MXVR_GADDR)

The `MXVR_ADDR` 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.

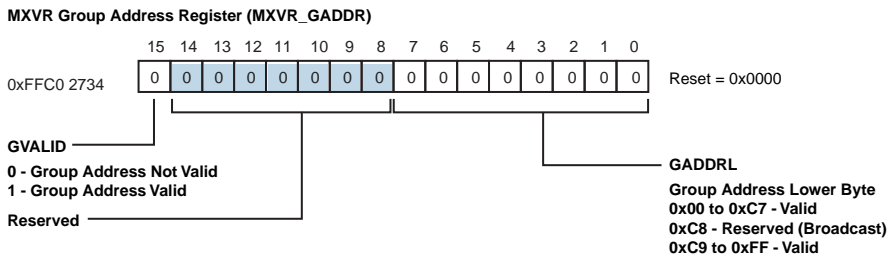


Figure 21-13. MXVR Group Address Register (MXVR_GADDR)

MXVR Alternate Address Register (MXVR_AADDR)

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. There is a `AVALID` bit which 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 `AADDR` field as the Alternate Address for checking the Destination Address field of incoming Asynchronous Packets.

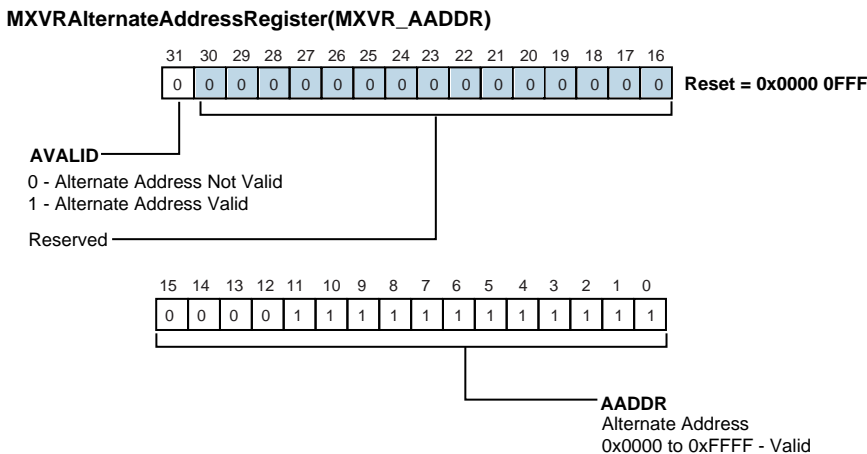


Figure 21-14. MXVR Alternate Address Register (MXVR_AADDR)

MXVR Allocation Table Registers (MXVR_ALLOC_0 – MXVR_ALLOC_14)

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).

MXVR Allocation Table Register 0 (MXVR_ALLOC_0)

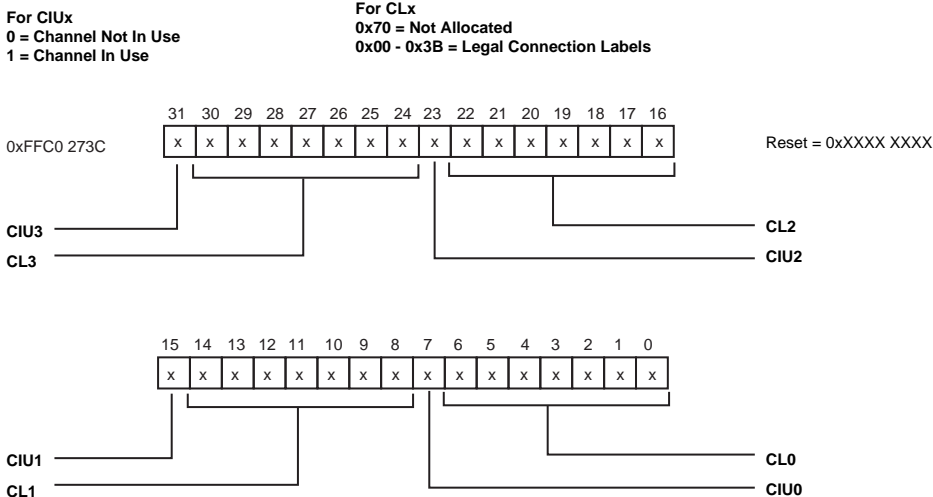


Figure 21-15. MXVR Allocation Table Register (MXVR_ALLOC_0)

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.

MXVR Registers

The Channel-In-Use (CIU_x) bit indicates whether a particular physical channel is “In-Use” by a node in the network. If the CIU_x bit is 0, physical channel x is not “In-Use”. If the CIU_x bit is 1, physical channel x is “In-Use”.

The Master node modifies its Allocation Table based on Allocate and 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 CIU_x 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 CIU_x bits only reflect which channels are “In-Use” by upstream nodes (nodes with lower $POSITION$ values).

MXVR Synchronous Logical Channel Assignment Registers (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7)

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 $MXVR_SYNC_LCHAN_0$ regis-

ter as an example of one of the Synchronous Logical Channel Assignment registers. All other Synchronous Logical Channel Assignment registers have the same format.

MXVR Synchronous Logical Channel Assignment Register 0 (MXVR_SYNC_LCHAN_0)

For LCHANPCx
 0000 - 0111 = Logical Channels 0 to 7
 1000 - 1110 = Reserved
 1111 = Unassigned

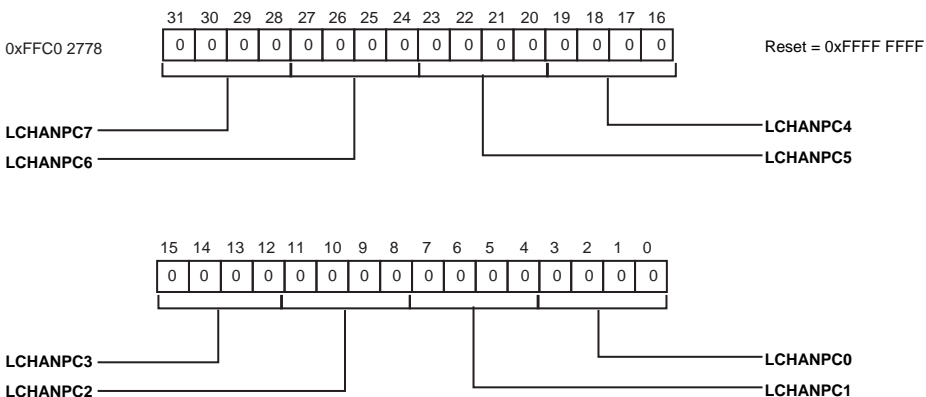


Figure 21-16. MXVR Synchronous Logical Channel Assignment Register (MXVR_SYNC_LCHAN_0)

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'd 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

MXVR Registers

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 Registers (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG)

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 MDMAENx 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.

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'd 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'd to L1 or L2 memory or which logical channel in the outgoing frame will be DMA'd from L1 or L2 memory and transmitted. The logical channels are defined in the `MXVR_SYNC_LCHANx` 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.

MXVR Registers

The DMA_x Bit-Swap Enable (BITSWAPEN_x) bit enables or disables bit swapping of the data that is DMA'd to and from L1 memory. If BITSWAPEN_x 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 DMA_x Two Byte Swap Enable (BY2SWAPEN_x) bit enables or disables two byte swapping of the data that is DMA'd to/from L1 or L2 memory. If BY2SWAPEN_x is set to 1, the data byte 0 will be swapped with data byte 1. If BY2SWAPEN_x 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_DMA_x_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 DMA_x Operation Flow (M_{FLOWx}) 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 Registers

`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 datas-

stream. 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 or until a DMA Error occurs. Note that DMA Enable/Disable always occur at the start 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

MXVR Registers

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 or until a DMA Error occurs. Note that DMA Enable/Disable always occur at the start 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 start 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 DMA Enable/Disable always occur at the start 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

MXVR Registers

“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’ed 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’ed 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 eighth 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

Media Transceiver Module (MXVR)

different frames. Note that the bytes of data between the “start pattern” and the transfer count, and the transfer count itself will be DMA’ed to L1 or L2 memory.

MXVR DMAx Configuration Register (MXVR_DMA0_CONFIG)

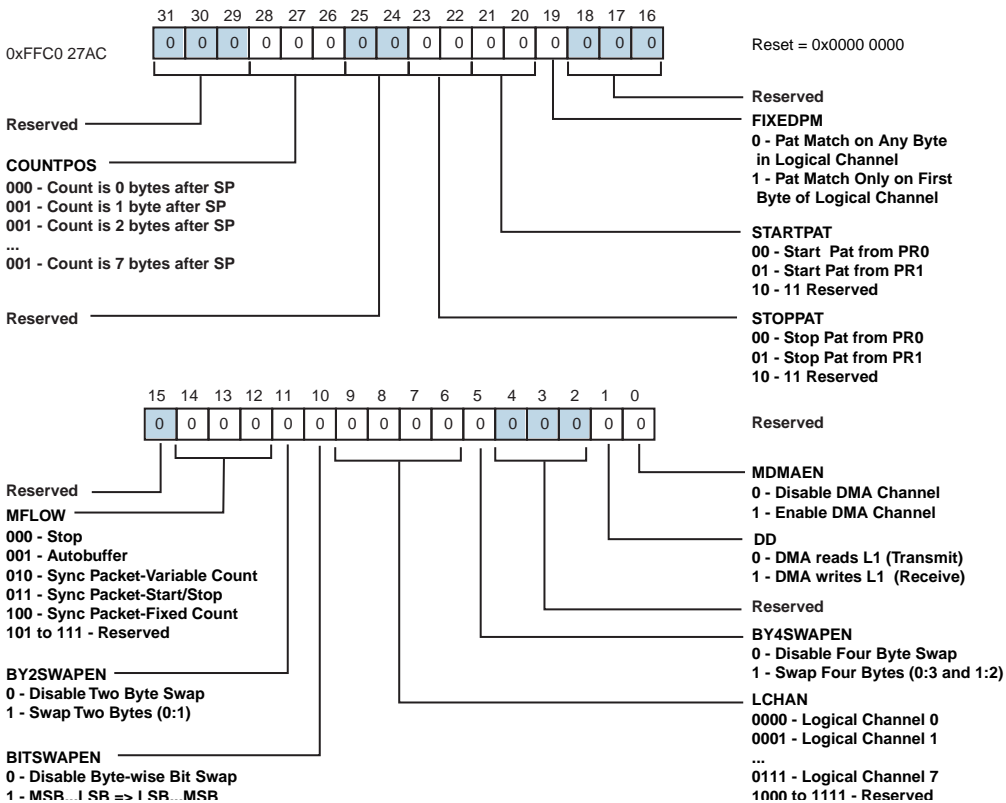


Figure 21-17. MXVR DMAx Configuration Register (MXVR_DMA0_CONFIG)

MXVR DMA Channel x Start Address Registers (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR)

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.

MXVR DMA Channelx Start Address Register (MXVR_DMA0_START_ADDR)

Bits 31-24 are fixed to 0xFF

All other bits are Read/Write when channel is disabled Read-Only when channel is enabled

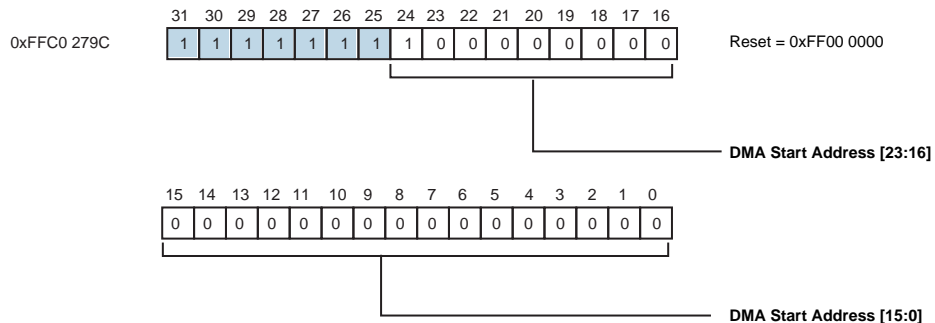


Figure 21-18. MXVR DMA Channel x Start Address Registers
(MXVR_DMAx_START_ADDR)

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

MXVR Registers

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 Registers (`MXVR_DMA0_CURR_ADDR` – `MXVR_DMA7_CURR_ADDR`)

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.

MXVR DMA Channel x Current Address Register (`MXVR_DMA0_CURR_ADDR`)

All bits are Read-Only

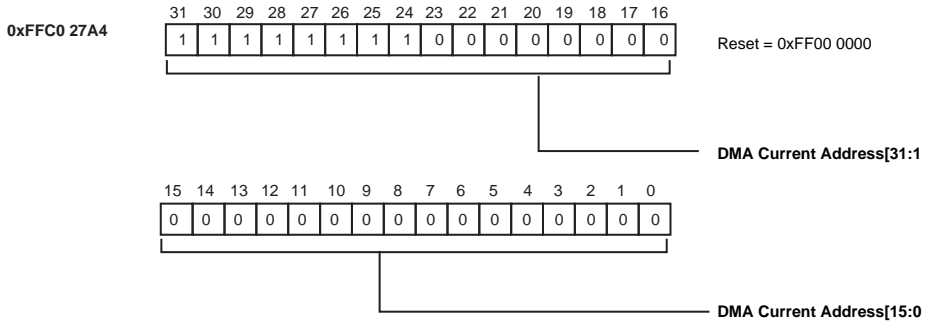


Figure 21-19. MXVR DMA Channel x Current Address Register (`MXVR_DMAx_CURR_ADDR`)

MXVR DMA Channel x Transfer Count Registers (`MXVR_DMA0_COUNT` – `MXVR_DMA7_COUNT`)

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 Registers

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.

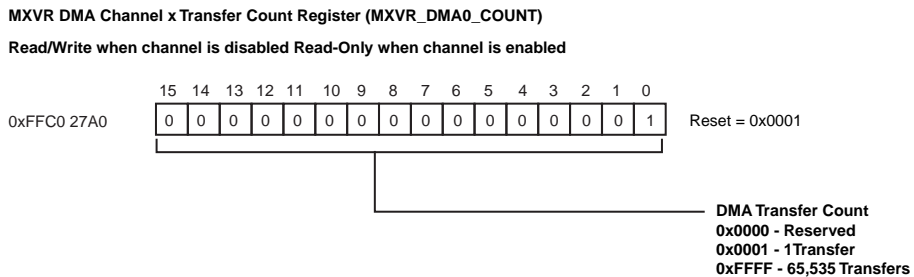


Figure 21-20. MXVR DMA Channel x Transfer Count Registers (MXVR_DMAx_COUNT)

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, eighth 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, eighth 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, eighth packet, etc.) will be written starting at `MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT`.

MXVR DMA Channel x Current Transfer Count Registers (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT)

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.

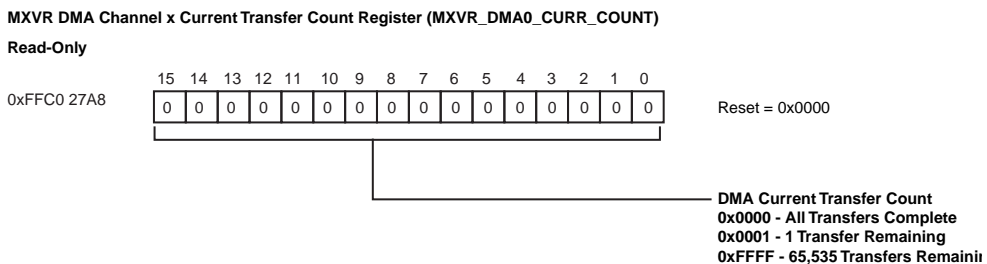


Figure 21-21. MXVR DMA Channel x Current Transfer Count Registers (MXVR_DMAx_CURR_COUNT)

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 Register (MXVR_AP_CTL)

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 (APT_B) and an Asynchronous Packet Receive Buffer (APR_B). The APR_B is capable of holding two received Asynchronous Packets.

MXVR Asynchronous Packet Control Register (MXVR_AP_CTL)

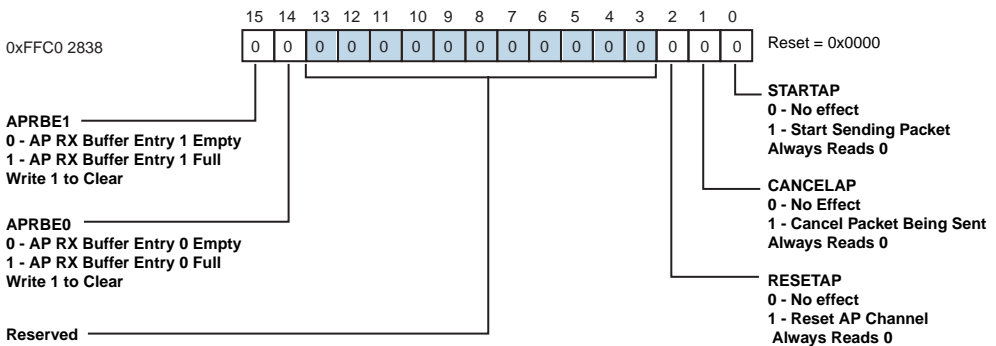


Figure 21-22. MXVR Asynchronous Packet Control Register (MXVR_AP_CTL)

The Start Asynchronous Packet Transmission (`STARTAP`) bit should be set to 1 once an asynchronous packet to be transmitted is written to the APT_B 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

MXVR Registers

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 Register (MXVR_APRB_START_ADDR)

The `MXVR_APRB_START_ADDR` register set 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

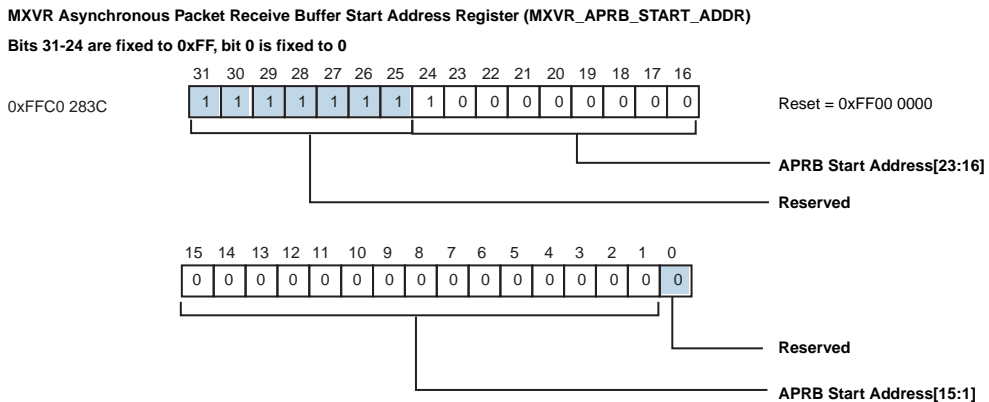


Figure 21-23. MXVR APRB Start Address Register (MXVR_APRB_START_ADDR)

MXVR Asynchronous Packet Receive Buffer Current Address Register (MXVR_APRB_CURR_ADDR)

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 1s and bit 0 will always be 0.

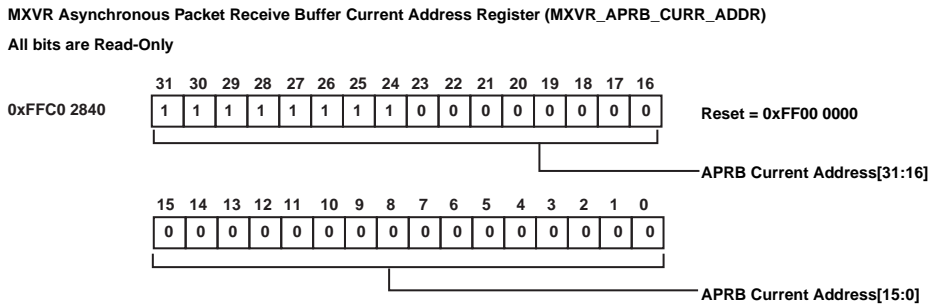


Figure 21-24. MXVR Asynchronous Packet Receive Buffer Current Address Register (MXVR_APRB_CURR_ADDR)

MXVR Asynchronous Packet Transmit Buffer Start Address Register (MXVR_APTB_START_ADDR)

The `MXVR_APTB_START_ADDR` registers set 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.

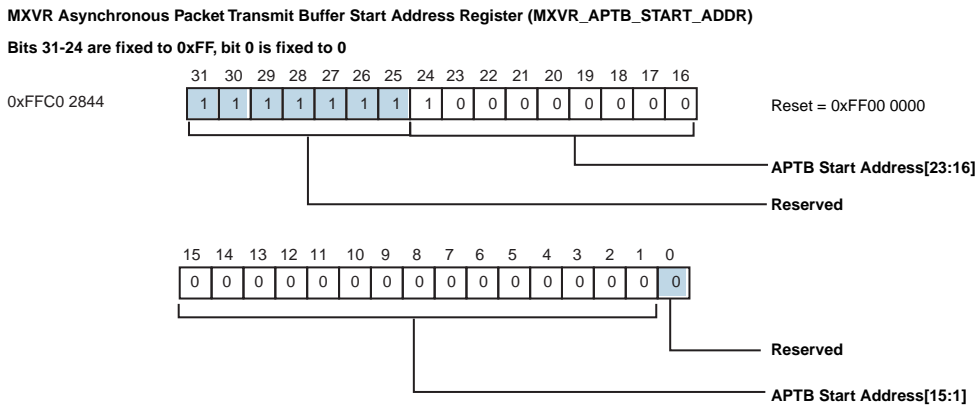


Figure 21-25. MXVR Asynchronous Packet Transmit Buffer Start Address Register (MXVR_APTB_START_ADDR)

MXVR Asynchronous Packet Transmit Buffer Current Address Register (MXVR_APTB_CURR_ADDR)

The `MXVR_APTB_CURR_ADDR` register is 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.

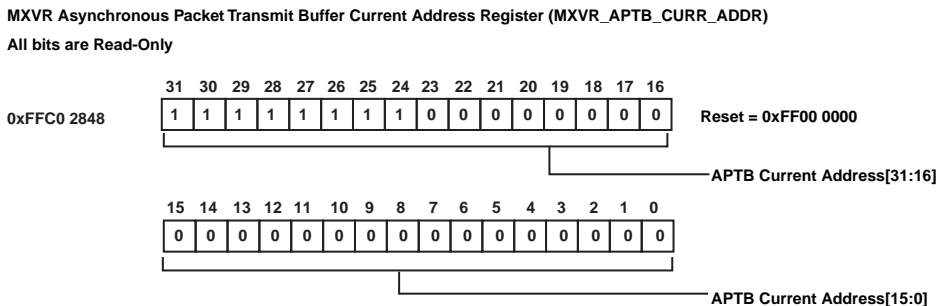


Figure 21-26. MXVR Asynchronous Packet Transmit Buffer Current Address Register (MXVR_APTB_CURR_ADDR)

MXVR Control Message Control Register (MXVR_CM_CTL)

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

MXVR Registers

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.

MXVR Control Message Control Register (MXVR_CM_CTL)

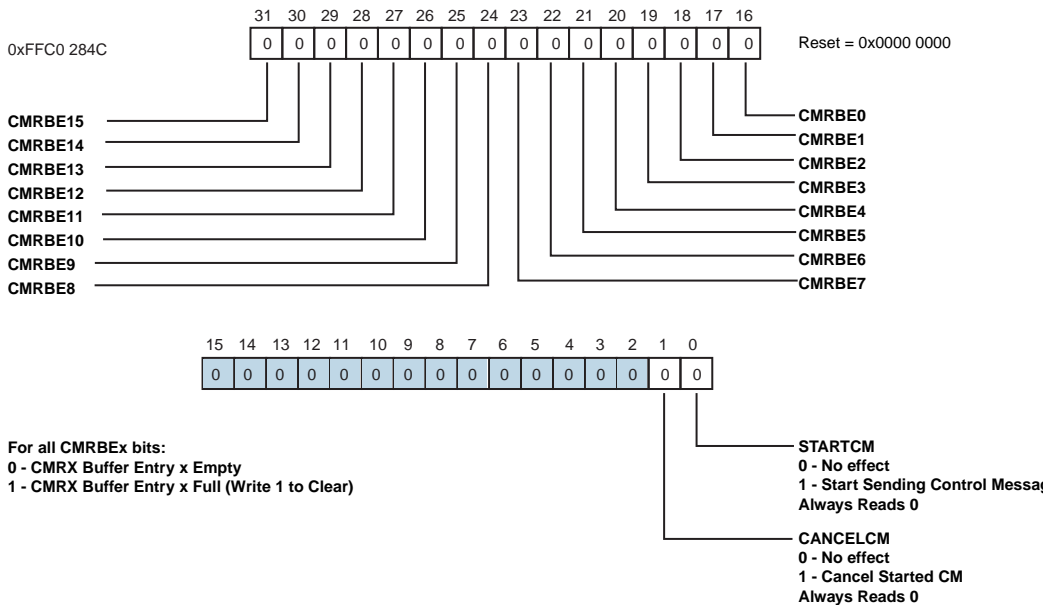


Figure 21-27. MXVR Control Message Control Register (MXVR_CM_CTL)

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 (normal or system) 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.

The Control Message Receive Buffer Entry x (`CMRBE x`) bits indicate whether entry x in the `CMRB` is full or empty. The `CMRBE x` 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 `CMRBE x` 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 `CMRBE x` 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 `CMRBE x` 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 Register (MXVR_CMRB_START_ADDR)

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.

MXVR Control Message Receive Buffer Start Address Register (MXVR_CMRB_START_ADDR)

Bits 31-24 are fixed to 0xFF, bit 0 is fixed to 0

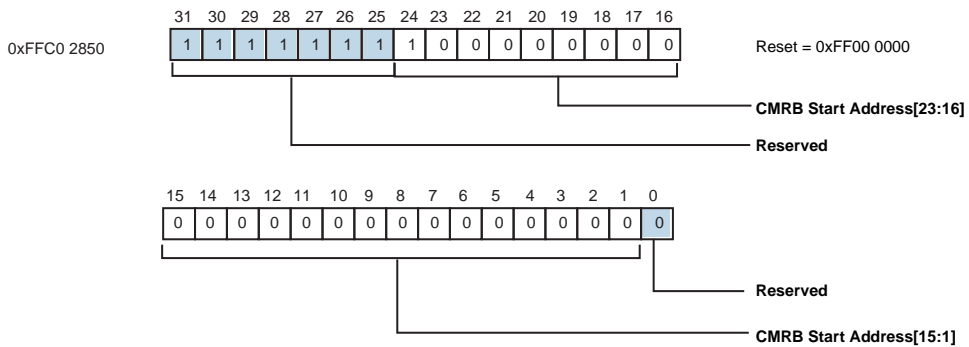


Figure 21-28. MXVR Control Message Receive Buffer Start Address Register (MXVR_CMRB_START_ADDR)

MXVR Control Message Receive Buffer Current Address Register (MXVR_CM RB_CURR_ADDR)

The `MXVR_CM RB_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 `CM RB`. The `CM RB` 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.

MXVR Control Message Receive Buffer Current Address Register (MXVR_CM RB_CURR_ADDR)

All bits are Read-Only

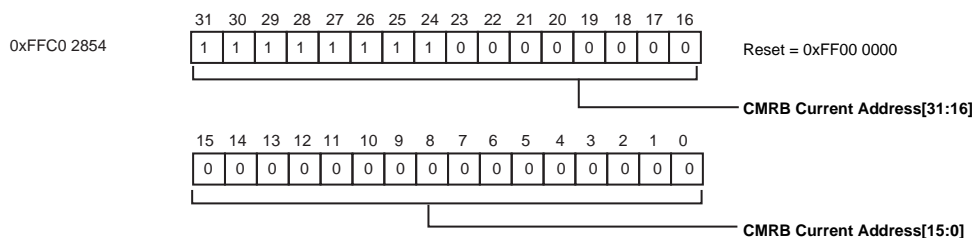


Figure 21-29. MXVR Control Message Receive Buffer Current Address Register (MXVR_CM RB_CURR_ADDR)

MXVR Control Message Transmit Buffer Start Address Register (MXVR_CMTB_START_ADDR)

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.

MXVR Control Message Transmit Buffer Start Address Register (MXVR_CMTB_START_ADDR)

Bits 31-24 are fixed to 0xFF, bit 0 is fixed to 0

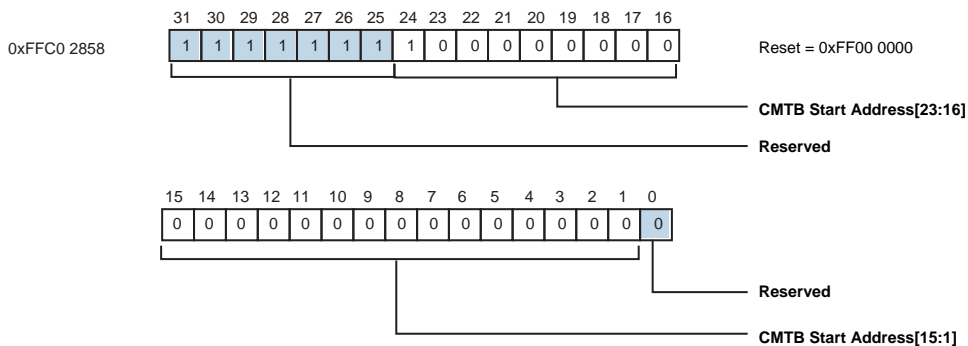


Figure 21-30. MXVR Control Message Transmit Buffer Start Address Registers (MXVR_CMTB_START_ADDR)

MXVR Control Message Transmit Buffer Current Address Register (MXVR_CMTB_CURR_ADDR)

The `MXVR_CMTB_CURR_ADDR` register is a read-only register which gives the current address that the Control Message Transmit DMA channel is reading from in the CMTB. The CMTB 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.

MXVR Control Message Transmit Buffer Current Address Register (MXVR_CMTB_CURR_ADDR)

All bits are Read-Only

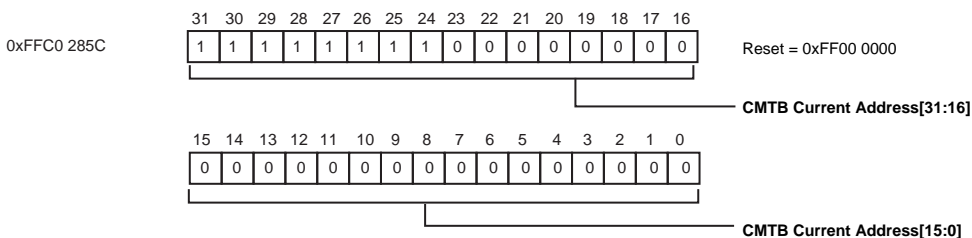


Figure 21-31. MXVR Control Message Transmit Buffer Current Address Register (MXVR_CMTB_CURR_ADDR)

MXVR Remote Read Buffer Start Address Register (MXVR_RRDB_START_ADDR)

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.

MXVR Remote Read Buffer Start Address Register (MXVR_RRDB_START_ADDR)

Bits 31–24 are fixed to 0xFF, bit 0 is fixed to 0

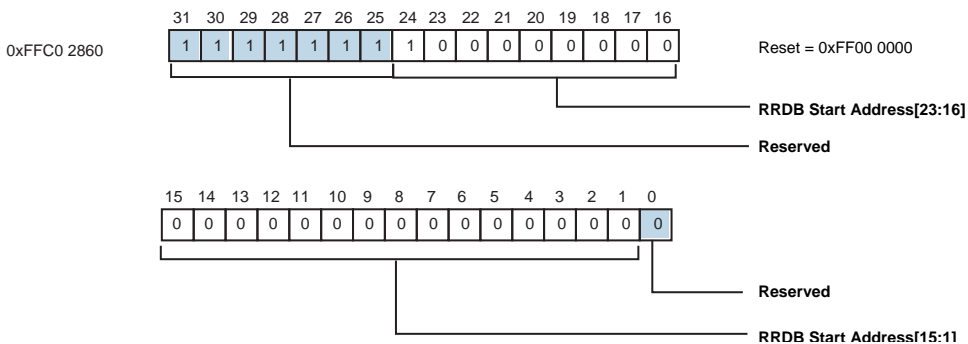


Figure 21-32. MXVR Remote Read Buffer Start Address Register (MXVR_RRDB_START_ADDR)

MXVR Remote Read Buffer Current Address Register (MXVR_RRDB_CURR_ADDR)

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.

MXVR Remote Read Buffer Current Address Register (MXVR_RRDB_CURR_ADDR)

All bits are Read-Only

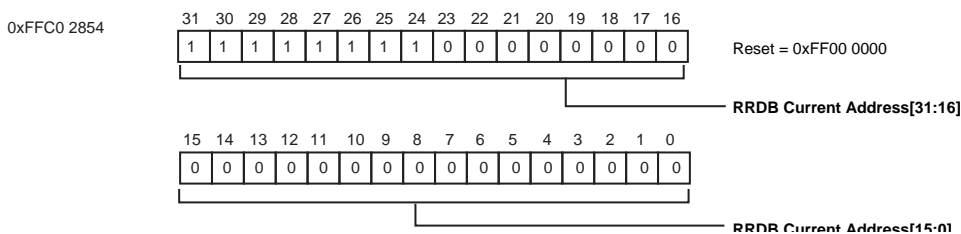


Figure 21-33. MXVR Remote Read Buffer Current Address Register (MXVR_RRDB_CURR_ADDR)

MXVR Pattern Registers

The MXVR has two sets of Pattern Registers: Pattern 0 Registers (`PRO`) 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 `PRO` or `PR1` and the “stop pattern” to be defined by either `PRO` 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 Registers (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1)

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.

MXVR Pattern Data Registers (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1)

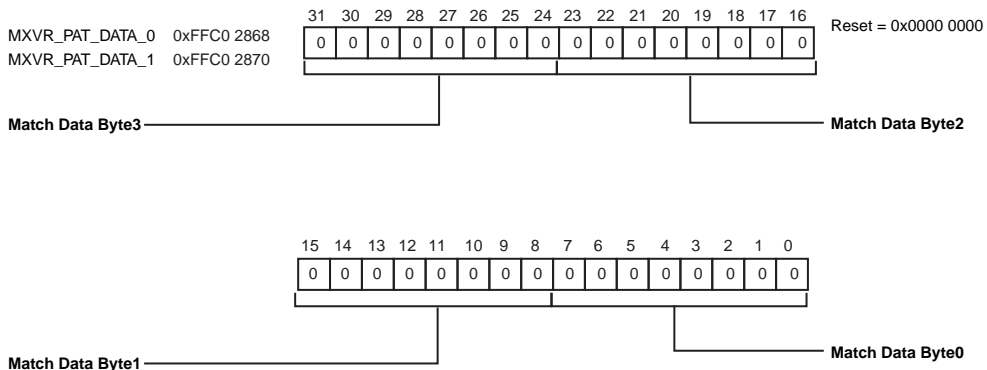


Figure 21-34. MXVR Pattern Data Registers (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1)

MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)

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.

MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)

For all bits

0 - Corresponding pattern data bit is not used in pattern matching

1 - Corresponding pattern data bit is used in pattern matching

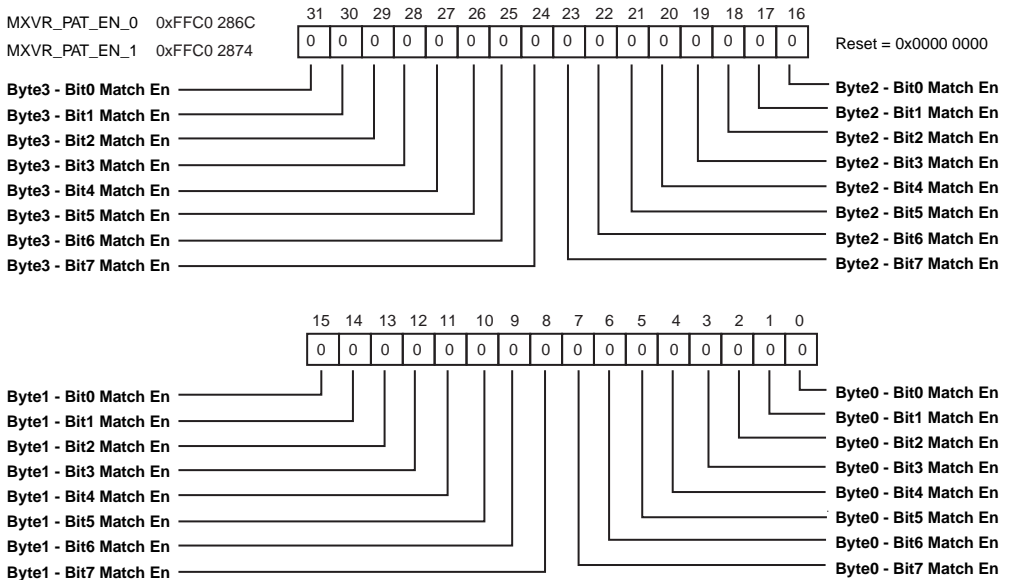


Figure 21-35. MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)

MXVR Registers

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.

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 Registers (`MXVR_FRAME_CNT_0`, `MXVR_FRAME_CNT_1`)

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 corre-

sponding 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.

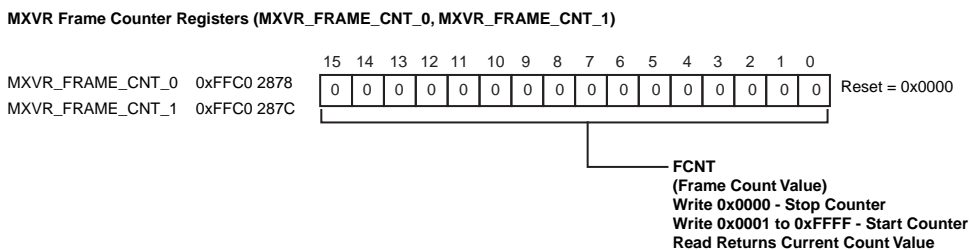


Figure 21-36. MXVR Frame Counter Registers (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1)

MXVR Routing Registers (MXVR_ROUTING_0 – MXVR_ROUTING_14)

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 addi-

MXVR Registers

tion, 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`.

MXVR Routing Register 0 (`MXVR_ROUTING_0`)

Write-only

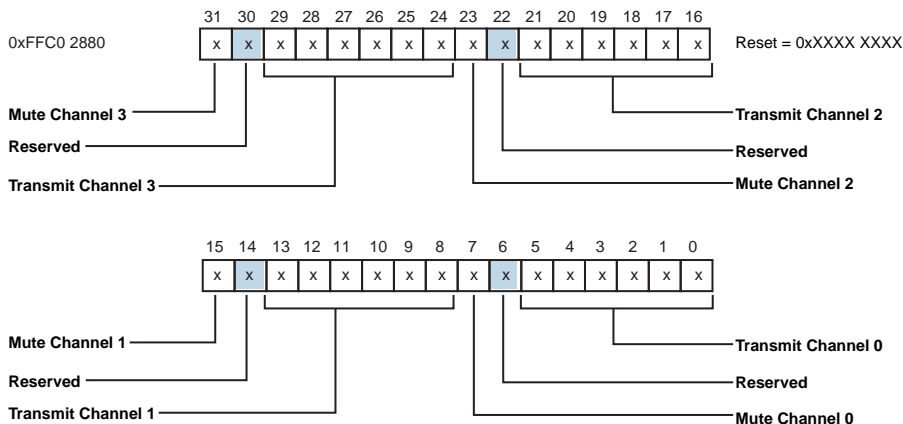


Figure 21-37. MXVR Routing Registers (`MXVR_ROUTING_0`)

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 21-37](#) 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 $4 * \text{RSB}$) 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 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 `0x00` can be transmitted on physical channel `0x01` and synchronous data received on physical channel `0x01` can be transmitted on physical channel `0x00` 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 `0x05` 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 Channel Mute 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 Channel Mute 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 Channel Mute entries for those channels will be active. For example, if physical channels `0x04` and `0x05` have the Channel Mute bit set, they will output `0x00` data. If a Logical Channel 0 is

MXVR Registers

defined as physical channels 0x04 and 0x05 and a synchronous data DMA channel is setup to transmit on Logical Channel 0, once the DMA channel is enabled the DMA'd data will be transmitted on physical channels 0x04 and 0x05. Once the synchronous data DMA channel is disabled, physical channels 0x04 and 0x05 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 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 Register (MXVR_BLOCK_CNT)

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 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 pream-

bles 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.

MXVR Block Counter Register (MXVR_BLOCK_CNT)

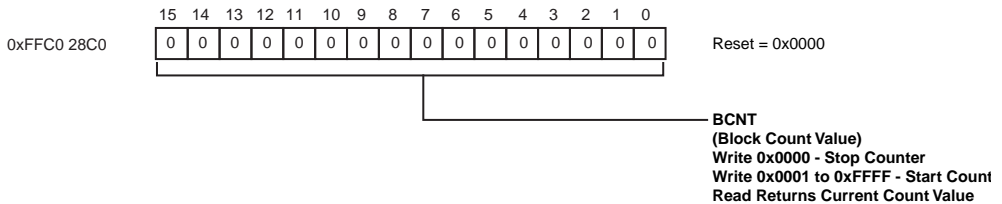


Figure 21-38. MXVR Block Counter Register (MXVR_BLOCK_CNT)

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 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 will pause 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 will change to 1 and the Status Change Interrupt will assert 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 clear 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 Registers

MXVR Clock Control Register (MXVR_CLK_CTL)

The MXVR_CLK_CTL register controls the MXVR Crystal Oscillator and the MXVR clock outputs.

MXVR Clock Control Register (MXVR_CLK_CTL)

R/W

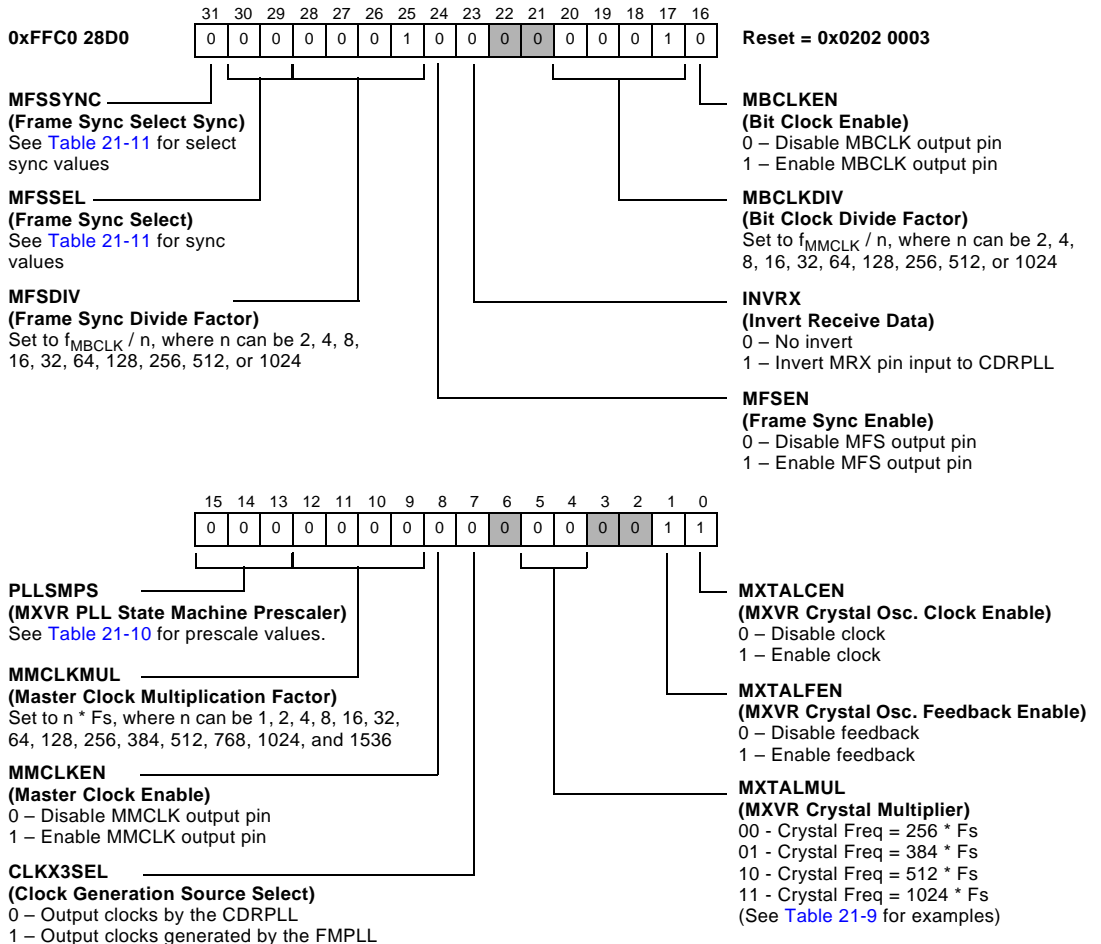


Figure 21-39. MXVR Clock Control Register (MXVR_CLK_CTL)

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 $256 * F_s$, $384 * F_s$, $512 * F_s$, or $1024 * F_s$. 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

MXVR Registers

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 ($1024 * F_s$). Table 21-9 shows all the crystal frequencies ($256 * F_s$, $384 * F_s$, $512 * F_s$, or $1024 * F_s$) 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 21-9. Crystal Input Frequencies

MXTALMUL	Multiply Factor	Crystal Frequency	Crystal Frequency Needed for Desired F_s		
			$F_s = 38$ kHz	$F_s = 44.1$ kHz	$F_s = 48$ kHz
b#00	8/2	$256 * F_s$	9.728 MHz	11.2896 MHz	12.288 MHz
b#01	8/3	$384 * F_s$	14.592 MHz	16.9344 MHz	16.432 MHz
b#10	8/4	$512 * F_s$	19.456 MHz	22.5792 MHz	24.576 MHz
b#11	8/8	$1024 * F_s$	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 (F_s). The frequency can be set to be $n * F_s$ 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 $1024 * F_s$ or $1536 * F_s$, the `MMCLK` duty cycle will be 50%. When `MMCLKMUL` is set to $1024 * F_s$ or $1536 * F_s$, the `MMCLK` duty cycle will be 33%. Note that the frequency of `MMCLK` 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 21-10](#) shows how the `PLLSMPS` field should be programmed based on the `SCLK` frequency.

Table 21-10. PLLSMPS Encoding Selection

SCLK Frequency Range	PLLSMPS
$116\text{MHz} < f_{\text{SCLK}} \leq 133\text{MHz}$	b#000
$99\text{MHz} < f_{\text{SCLK}} \leq 116\text{MHz}$	b#001
$83\text{MHz} < f_{\text{SCLK}} \leq 99\text{MHz}$	b#010
$66\text{MHz} < f_{\text{SCLK}} \leq 83\text{MHz}$	b#011
$49\text{MHz} < f_{\text{SCLK}} \leq 66\text{MHz}$	b#100
$33\text{MHz} < f_{\text{SCLK}} \leq 49\text{MHz}$	b#101
$16\text{MHz} < f_{\text{SCLK}} \leq 33\text{MHz}$	b#110
$f_{\text{SCLK}} \leq 16\text{MHz}$	b#111

MXVR Registers

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 $1024 * F_s$ or $1536 * F_s$, the rising edge of MBCLK occurs in sync with the rising edge of MMCLK. When MMCLKMUL is set to $1024 * F_s$ or $1536 * F_s$, the rising edge of MBCLK occurs in sync with the falling edge of MMCLK. Note that the frequency of MBCLK 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 MMCKEN, MBCKEN, and MFSEN are all set to 0.

The Frame Sync Select (MFSEL) 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 MFSEL field is set to b#00, the MFS will generate a 50% duty cycle clock. If the MFSEL 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 MFSEL 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 MFSEL should only be changed when MMCKEN, MBCKEN, 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 (MFSEL = b#00) or an active-high pulse (MFSEL = 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 (MFSEL = 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

MXVR Registers

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 21-11. 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 Register (MXVR_CDRPLL_CTL)

MXVR Clock/Data Recovery PLL Control Register (MXVR_CDRPLL_CTL)

R/W

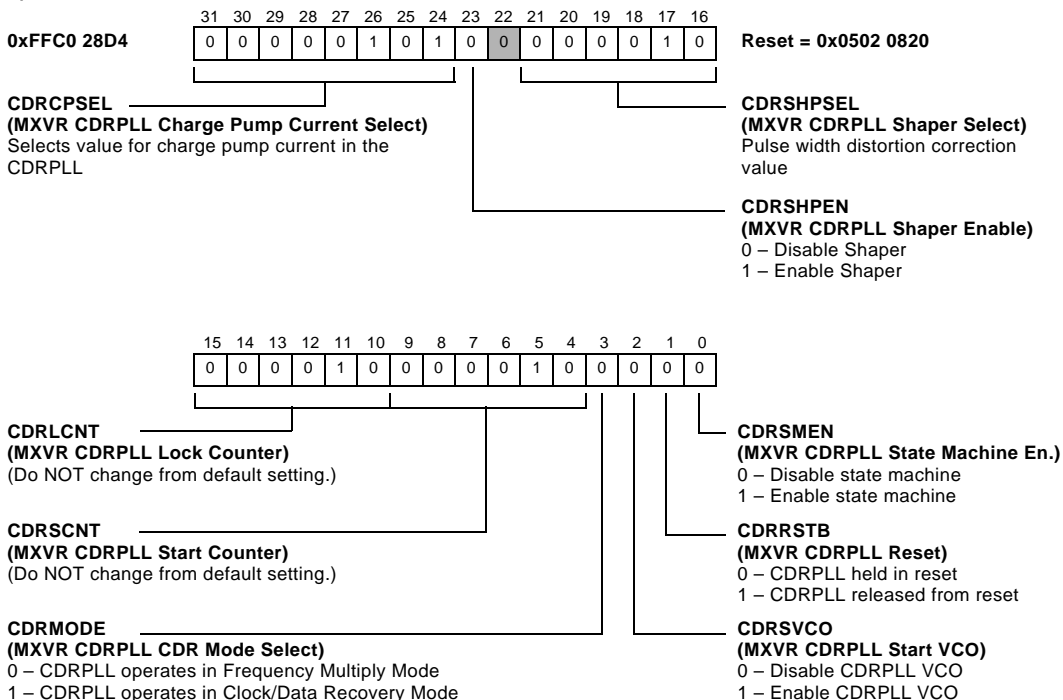


Figure 21-40. MXVR Clock/Data Recovery PLL Control Register (MXVR_CDRPLL_CTL)

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. The CDRSMEN bit is set to 0 by reset.

MXVR Registers

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 `CDRSVCO` 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.

Media Transceiver Module (MXVR)

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.

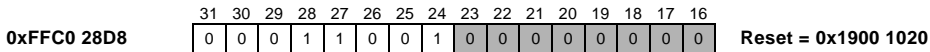
The MXVR Clock/Data Recovery PLL Shaper Enable (CDRSHPEN) bit enables or disables the CDRPLL Shaper which corrects pulse width distortion in the incoming data stream. The CDRSHPEN bit is set to 0 by reset.

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.

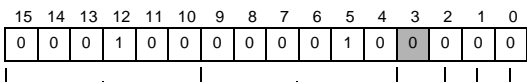
MXVR Frequency Multiply PLL Control Register (MXVR_FMPLL_CTL)

MXVR Frequency Multiply PLL Control Register (MXVR_FMPLL_CTL)

R/W



FMCPSEL
(MXVR FMPLL Charge Pump Current Select)
 Selects value for charge pump current in the FMPLL



FMLCNT
(MXVR FMPLL Lock Counter)
 (Do NOT change from default setting.)

FMSCNT
(MXVR FMPLL Start Counter)
 (Do NOT change from default setting.)

FMSMEN
(MXVR FMPLL State Machine Enable)
 0 – Disable state machine
 1 – Enable state machine

FMRSTB
(MXVR FMPLL Reset)
 0 – FMPLL held in reset
 1 – FMPLL released from reset

FMSVCO
(MXVR FMPLL Start VCO)
 0 – Disable FMPLL VCO
 1 – Enable FMPLL VCO

Figure 21-41. MXVR Frequency Multiply PLL Control Register (MXVR_FMPLL_CTL)

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. 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 `FMSV0` 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 Registers

MXVR Pin Control Register (MXVR_PIN_CTL)

MXVR Pin Control Register (MXVR_PIN_CTL)

R/W

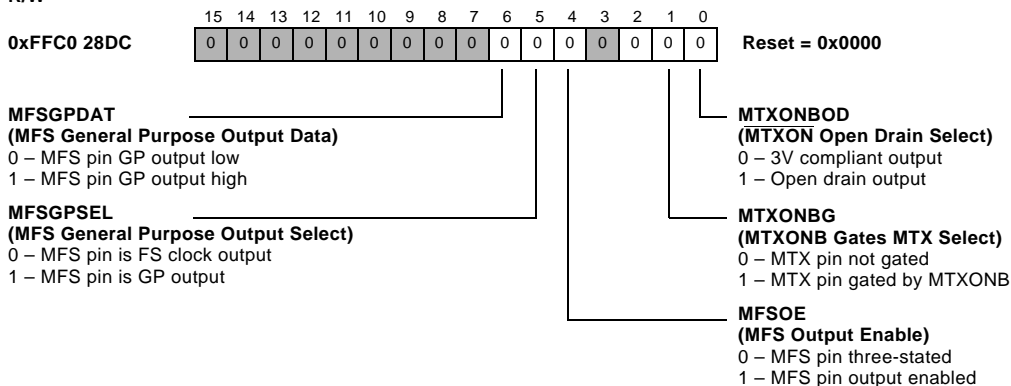


Figure 21-42. MXVR Pin Control Register (MXVR_PIN_CTL)

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 Register (`MXVR_SCLK_CNT`)

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

MXVR Registers

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).

MXVR System Clock Count Register (MXVR_SCLK_CNT)

R/W

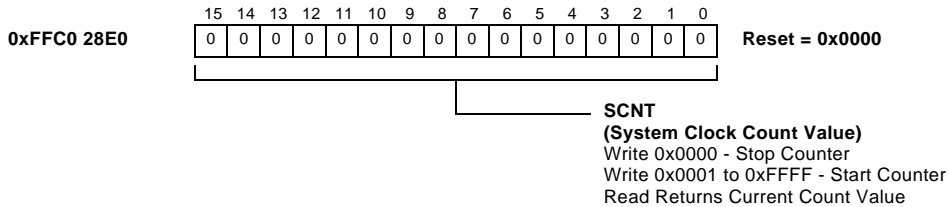


Figure 21-43. MXVR System Clock Count Register (MXVR_SCLK_CNT)

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 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 MXVR master node will be triggered to start up the network based on an external event (for example, car ignition, power switch, etc.), while an ADSP-BF54x 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 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

General Operation

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 set to 0 and when the Receive PHY detects network activity, the status output is set to 1.

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 from the hibernate state if the MXVRWE bit in the VR_CTL register is set to 1. For more information on the VR_CTL register, see the “Dynamic Power Management” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*. 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 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 to enter a low-power state.

The third method for network activity detection is handled completely outside of the ADSP-BF54x. In this method the Receive PHY status output controls the power supply for the ADSP-BF54x. When the Receive PHY status output indicates that there is no network activity, the power supply for the ADSP-BF54x is gated off. When the Receive PHY status output indicates that there is network activity, the power supply for the ADSP-BF54x is turned on. Once the reset to the ADSP-BF54x 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 pin multiplexing, the MXVR_CONFIG register, the MXVR_CLK_CTL register, the MXVR_ROUTING_x registers, and the buffer state 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.

General Operation

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. See the “General-Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)* for details on how the GPIO registers should be programmed.

Master mode initialization of the MXVR_CONFIG register

The `MXVREN` bit should remain 0 (keeping the MXVR disabled) until the MXVR PLL is 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 the 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 MX0 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;
```

General Operation

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.

Enabling the MXVR PLLs

{Description TBD.}

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 be 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, MFSEL, and MFSSYNC fields in the MXVR_CLK_CTL register to define the frequency and relationship of the MXVR output clocks.
4. Wait 1 μ sec.
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

Summary description TBD.

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

General Operation

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 `GADDR` 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_GADDR` 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, Muting, and 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. The MXVR can mute individual physical channels and can automatically mute physical channels when a DMA channel has completed transmission. The MXVR can also route incoming synchronous data from one physical channel to one or more outgoing physical channels based on the Routing registers when 2 frames of synchronous data delay is selected. For an MXVR Master node, incoming synchronous data (on physical channels which the MXVR is not transmitting synchronous data on) will always be routed with 2 frames of delay. For an MXVR Slave node, incoming synchronous data (on physical channels which the MXVR is not transmitting data on) can either be routed

General Operation

with 2 frame delays or with 0 frame delays. In addition, the MXVR Slave node is capable of transmitting synchronous data on any physical channel regardless of whether 2 frames of delay or 0 frames of delay is selected.

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.

In order to set up a DMA channel for 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. 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 which data will be transmitted in 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 bits in the `MXVR_DMAx_CONFIG` register should be programmed. The `MDMAENx` bit should be set to 0 until the DMA channel is completely programmed, 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.

General Operation

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.

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.

Synchronous Data Reception

DMA

Pattern Matching

Asynchronous Packet Transmission

The MXVR Asynchronous Packet Transmit Buffer (APT_B) is an area of memory that is allocated to hold an asynchronous packet to be transmitted. The APT_B must reside in L1 or L2 memory and the starting address of the APT_B 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 APT_B must be 1026 bytes.

Once the asynchronous packet to be transmitted is written to the APT_B, 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 APT_B 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'd 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 21-12](#).

Table 21-12. 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)

General Operation

Table 21-12. Asynchronous Packet Transmit Buffer Field Offsets

APT B Address Offsets	Field Name
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.

General Operation

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 21-13](#).

Table 21-13. Asynchronous Packet Receive Buffer Entry Offsets

APRB Entry Offset	APRB Entry Number
<i>MXVR_APRB_START_ADDR</i> + 0x000	AP Receive Buffer Entry 0
<i>MXVR_APRB_START_ADDR</i> + 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 21-14](#). 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 21-14. 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

Table 21-14. Asynchronous Packet Receive Buffer Entry Field Offsets

APRB Entry Address Offsets	Field Name
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

General Operation

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 Get Source. 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.

Media Transceiver Module (MXVR)

The `CM Message Type` field determines which type of control message is being sent. [Table 21-15](#) 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 21-15. 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	Allocate Control Message
0x04	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 take DMA the `Transmission Status` that was received back from the destination over the bus into the `CM Transmission Status` field in the `CMTB`. [Table 21-16](#) gives the meaning of the transmission status values received back when single cast addressing is used.

Table 21-16. 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

General Operation

Table 21-17 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 21-17. 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
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

Table 21-17. Group cast/Broadcast Transmission Status Encodings

CM Transmission Status	Meaning of Transmission Status
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 21-18](#).

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

General Operation

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'ed 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'ed 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 21-18](#).

Table 21-18. 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 nodes 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 nodes’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 21-19](#).

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

General Operation

received back by the MXVR and is DMA'ed 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 21-19](#).

Table 21-19. 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
0x07	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 21-20](#).

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,

General Operation

CM Write Length and CM Write Data fields are DMA'ed 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'ed 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 21-20](#).

Table 21-20. 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
0x07	CM Write Length
0x0A - 0x11	CM Write Data
0x12 - 0x13	Reserved
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 21-21](#).

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.

General Operation

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'ed 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'ed 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 21-21](#).

Table 21-21. 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
0x06	CM Message Type (Write 0x03)
0x07	Reserved (Write 0x00)
0x08	CM Allocate Number Requested
0x07	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 21-22](#) 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 21-22. 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.

General Operation

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`

Media Transceiver Module (MXVR)

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 21-23](#).

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:

$$\text{Uppermost Synchronous Channel Number} = (4 * \text{RSB}) - 1$$

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 21-23](#).

Table 21-23. Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address

General Operation

Table 21-23. Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x04	CM Source Address
0x06	CM Message Type (Write 0x03)
0x07	Reserved (Write 0x00)
0x08	CM De-Allocate Connection Label
0x07	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 21-24](#) 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 21-24. 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 Get Source Control Message Transmission

The remote get source control message is used to determine which node is transmitting data on a particular physical channel. A remote get source control message can be sent using broadcast addressing and the node which is transmitting on the channel specified in the CM Get Source Channel field will respond with its physical address, logical address and group address. In addition, by setting the CM Get Source Channel field to 0xFF and sending a remote get source 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 Desti-

General Operation

nation Address, CM Source Address, CM Message Type (0x04), and CM Get Source Channel fields should be written to the CMTB at the address offsets given in [Table 21-25](#).

For remote get source 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 Get Source Channel field.

The CM Get Source Channel field contains a physical channel number. The CM Get Source 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:

$$\text{Uppermost Synchronous Channel Number} = (4 * \text{RSB}) - 1$$

Once a remote get source 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 Get Source Channel fields are DMA'ed from the CMTB to the MXVR. Once the MXVR wins arbitration the remote get source 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'ed back to the CMTB. The response will be stored in the CM Get Source Physical Address (Low), CM Get Source Group Address (Low), CM Get Source Logical Address

Media Transceiver Module (MXVR)

(Low) and CM Get Source 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 21-25](#).

Table 21-25. 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 0x03)
0x07	Reserved (Write 0x00)
0x08	CM GetSource Channel
0x07	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

General Operation

If the destination node is an MXVR, the destination node will respond to the remote get source control message if the destination node is routing data onto or is muting the channel specified in the CM Get Source Channel field or if the CM Get Source Channel field is 0xFF. Other types of transceivers respond when the specified channel is routed.

If the destination node responds to the remote get source control message, the node will return the low byte of its physical address (POSITION) in the CM Get Source Physical Address (Low) field.

If the destination node responds to the remote get source control message, the node will return the low byte of its group address (GADDR) in the CM Get Source Group Address (Low) field.

If the destination node responds to the remote get source control message, the node will return its logical address (LADDR) in the CM Get Source Logical Address (High) and CM Get Source Logical Address (Low) fields.

If the remote get source 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 get source 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 get source 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 get source 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.

General Operation

The 16 CMRB Entries are stored as address offsets to the CMRB start address programmed in the `MXVR_CM RB_START_ADDR` register. The address offsets for the 16 CMRB Entries are given in [Table 21-26](#).

Table 21-26. Control Message Receive Buffer Entry Offsets

CMRB Entry Offset	CMRB Entry Number
<code>MXVR_CM RB_START_ADDR + 0x000</code>	CM Receive Buffer Entry 0
<code>MXVR_CM RB_START_ADDR + 0x016</code>	CM Receive Buffer Entry 1
<code>MXVR_CM RB_START_ADDR + 0x02C</code>	CM Receive Buffer Entry 2
<code>MXVR_CM RB_START_ADDR + 0x16 * x</code>	CM Receive Buffer Entry x
<code>MXVR_CM RB_START_ADDR + 0x14A</code>	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 21-27](#).

Table 21-27. 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 over the network. When a remote read control message is received by the MXVR, the 8 bytes of data requested will be DMA'ed 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'ed 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.

General Operation

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 21-28](#) gives the offsets of the RRDB Data, the RRDB Write Address, and the RRDB Write Length fields in the RRDB.

Table 21-28. 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 mes-

sage, 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”.

General Operation

Remote Get Source Reception

The reception of Remote Get Source control messages by the MXVR is handled completely in hardware. No software intervention is required.

If a Remote Get Source control message is received by the MXVR, the RGSIP bit in the MXVR_STATE_0 register will change to 1 to indicate a Remote Get Source control message is being processed. The MXVR will first determine whether it should respond to the Remote Get Source control message. The MXVR will respond if the MXVR is muting or routing data onto the channel specified in the CM Get Source Channel field or if the CM Get Source 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* bit in the appropriate MXVR_ROUTING_x register is set to any value other than *n*. If the CM Get Source 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 Get Source Channel field has a value between $4 * RSB$ and 0xFE.

When the MXVR responds to a Remote Get Source 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 Get Source 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 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 fre-

quencies, and controlling the operating mode of the core and the system PLL. See the “Dynamic Power Management” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*. Within the ADSP-BF54x dynamic power management framework, the MXVR has six general power/functionality states as shown in Table 21-29.

Table 21-29. ADSP-BF54x Power/MXVR Functionality States

ADSP-BF54x 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, <u>MRXON</u> , CANRX	Not Active
Power Gated Off to ADSP-BF54x	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.

General Operation

Full On Mode

When the ADSP-BF54x 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-BF54x Blackfin 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 MX0, 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 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-BF54x *Blackfin 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

General Operation

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 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 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 `SIC_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

General Operation

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 MX0, 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 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 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 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 to exit Deep Sleep Mode and begin the hardware reset booting sequence.

If a crystal is connected between MXI and MX0, 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 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 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 is in Hibernate State.

There are four wake-up sources that can wake the ADSP-BF54x 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 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 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 asser-

General Operation

tion 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 will begin the hardware reset booting sequence.

If a crystal is connected between MXI and MX0, 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 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

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 should be gated off and pulled to 0V. This effectively reduces the ADSP-BF54x power consumption to zero. Typically the MOST FOR status output would be used to gate the ADSP-BF54x power supplies on and off based on the reception of modulated light.

22 KEYPAD INTERFACE

This chapter describes the 16-pin programmable keypad interface and includes the following sections:

- [“Interface Overview” on page 22-1](#)
- [“Description of Operation” on page 22-2](#)
- [“Functional Description” on page 22-7](#)
- [“Programming Model” on page 22-9](#)
- [“Keypad Registers” on page 22-10](#)
- [“Programming Examples” on page 22-22](#)

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

Description of Operation

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

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 22-1](#).

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.

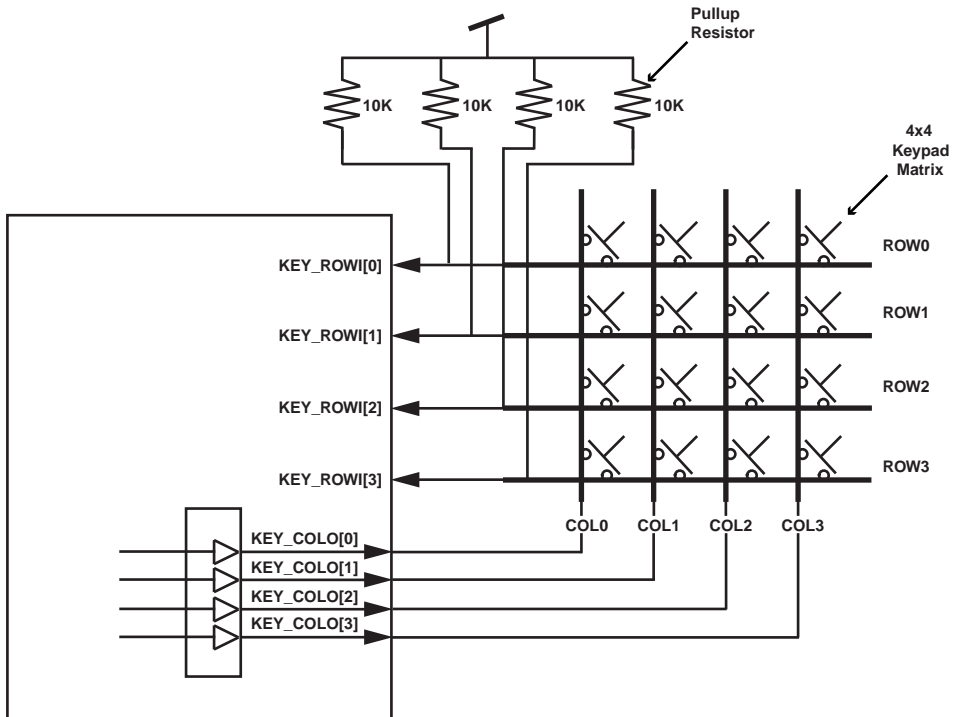


Figure 22-1. ADSP-BF54x Blackfin Processor Keypad Interface

Description of Operation

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 $(\text{KPAD_ROWEN} + 1)$ determines the number of rows enabled in the input keypad matrix. Similarly, the value of the $(\text{KPAD_COLEN} + 1)$ determines the number of active columns.

Table 22-1. 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 details about hibernation and GPIO, see the “Dynamic Power Management” and “General-Purpose Ports” chapter of the ADSP-BF4x Blackfin Processor Hardware Reference (Volume 1 of 2).

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 (T_{db}) and column drive period (T_{CW}).

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 T_{db} . 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 T_{CW} . With external pull-up resistors pulling up all row inputs, the Keypad interface logic pulls down one column at a time for T_{CW} 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

Description of Operation

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 22-2](#) 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

Functional Description

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.

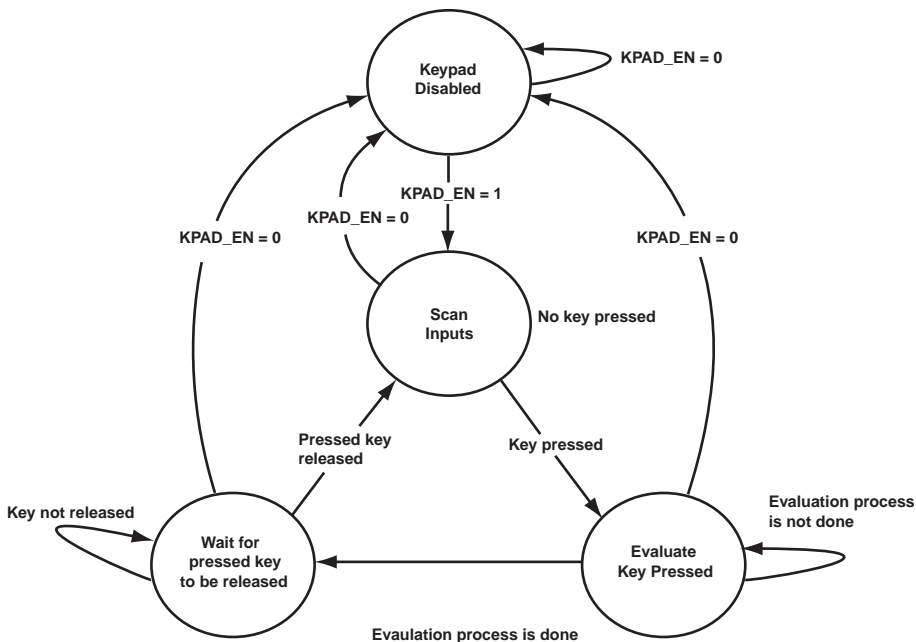


Figure 22-2. State Diagram

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 22-2. Control/Status/Data Registers

Name	Address Offset	Access	Description
KPAD_CTL	0xFFC04100	R/W	Keypad control register on page 22-10
KPAD_PRESCALE	0xFFC04104	R/W	Keypad prescale register on page 22-14
KPAD_MSEL	0xFFC04108	R/W	Keypad multiplier select register on page 22-15
KPAD_ROWCOL	0xFFC0410c	R/WC	Keypad row-column register on page 22-16
KPAD_STAT	0xFFC04110	R/W1C	Keypad status register on page 22-19
KPAD_SOFTEVAL	0xFFC04114	R/W	Keypad software evaluate register on page 22-21

Keypad Control Register (KPAD_CTL)

The keypad control (KPAD_CTL) register, shown in [Figure 22-3](#) and [Table 22-3](#) 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.

Keypad Control Register (KPAD_CTL)

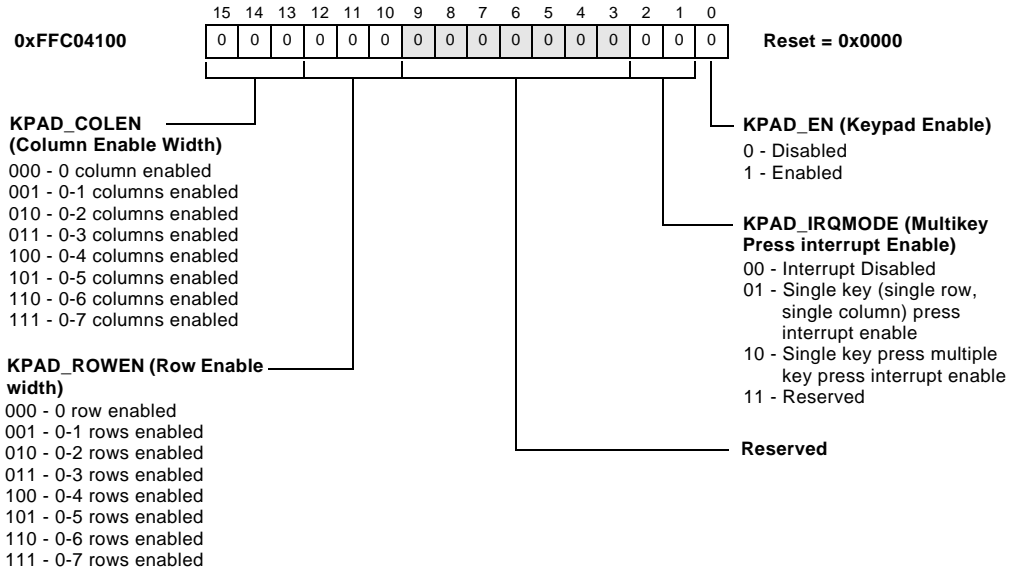


Figure 22-3. Keypad Control Register (KPAD_CTL)

Keypad Registers

Table 22-3. 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.
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

Table 22-3. Keypad Control Register Bit Descriptions (Cont'd)

Item	Bit(s)	Value	Description
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 Registers

KPAD_PRESCALE Register

The KPAD_PRESCALE register, shown in Figure 22-4, is used to program the pre-scale value that would be used in deriving delay parameters that the interface module should be sensitive to.

Keypad Prescale Register (KPAD_PRESCALE)

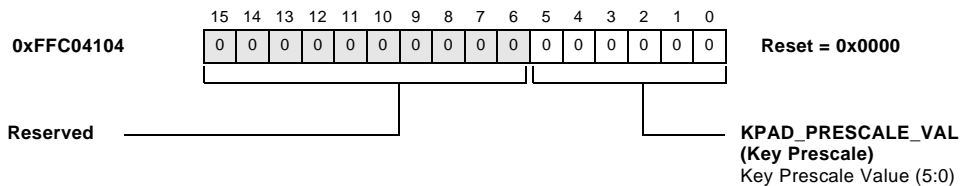


Figure 22-4. Key Prescale Register (KPAD_PRESCALE)

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 (T_{db}) and column drive period (T_{cw}).

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:

$$\text{Prescale Value} = \frac{\langle \text{SCLK Frequency} \rangle \times T}{1024} - 1$$

When using this formula, please note the unit of `SCLK` and the time scale must agree. For example, to generate a 0.1 ms timescale, the formula is:

$$\text{Prescale Value} = \frac{\langle \text{SCLK Frequency in MHz} \rangle \times 100 \mu\text{s}}{1024} - 1$$

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.

KPAD_MSEL Register

The `KPAD_MSEL` register, shown in [Figure 22-5](#), 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:

$$\text{COLDRV_SCALE} = [(T_{cw} * SCLK) / (KPAD_PRESCALE + 1) * 1024] - 1$$

$$\text{DBON_SCALE} = [(T_{db} * SCLK) / (KPAD_PRESCALE + 1) * 1024] - 1$$

Keypad Registers

Keypad Multiplier Select Register (KPAD_MSEL)

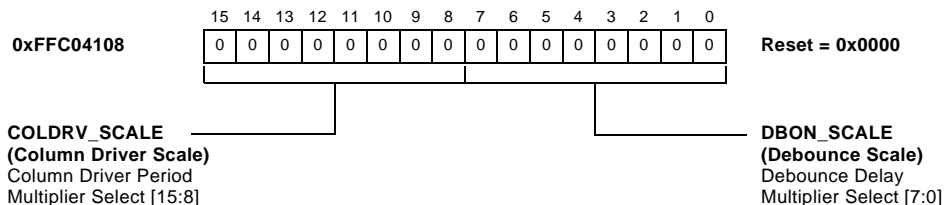


Figure 22-5. Keypad Multiplier Select Register (KPAD_MSEL)

KPAD_ROWCOL Register

The KPAD_ROWCOL register, shown in [Figure 22-6](#), is used to register the input row values and column output values once the Interface logic gets to a valid state.

Keypad Row-Column Register (KPAD_ROWCOL)

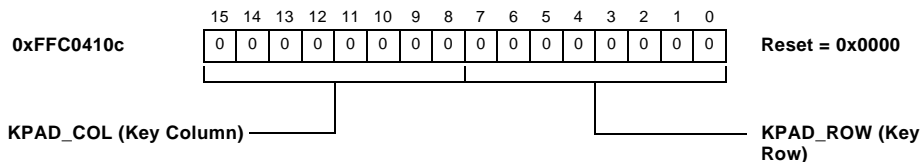


Figure 22-6. Keypad Row-Column Register (KPAD_ROWCOL)

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.

Keypad Registers

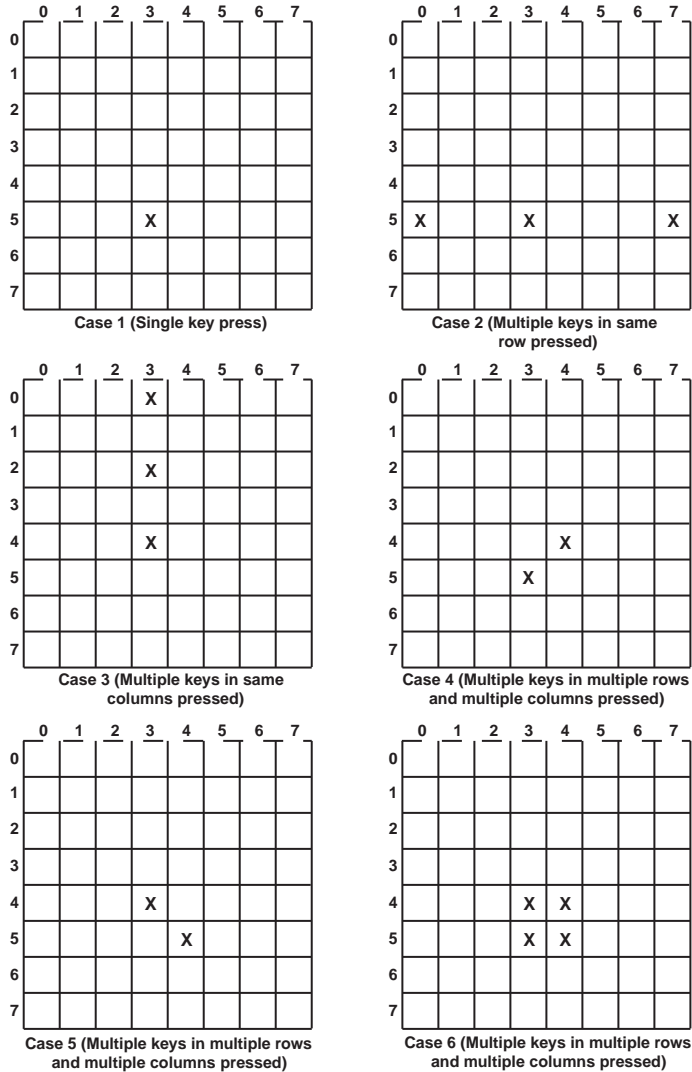


Figure 22-7. Interrupt Generation When X-Key Pressed

Figure 22-7 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.

KPAD_STAT Register

The `KPAD_STAT` register, shown in Figure 22-8, 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.

Keypad Registers

Keypad Status Register (KPAD_STAT)

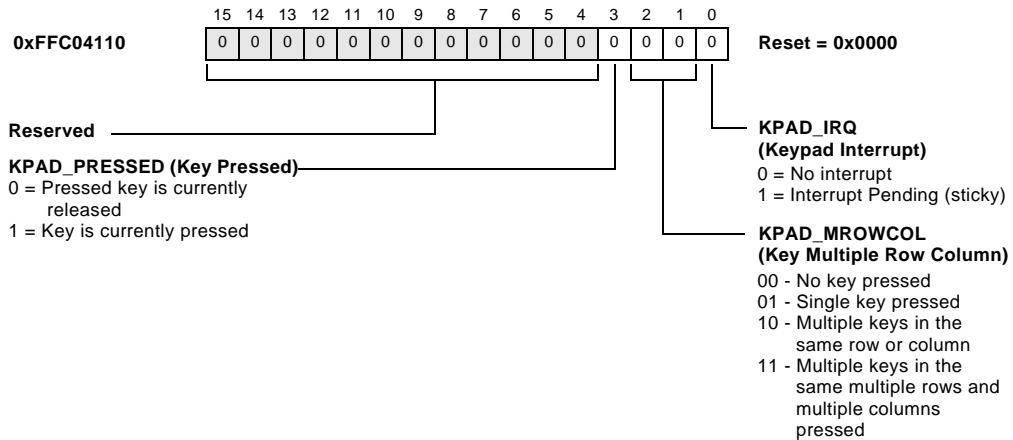


Figure 22-8. Keypad Status Register (KPAD_STAT)

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 routine 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`.

KPAD_SOFTEVAL Register

The `KPAD_SOFTEVAL` register, shown in [Figure 22-9](#), is used to force the interface into the evaluate state.

Keypad Software Evaluate Register (KPAD_SOFTEVAL)

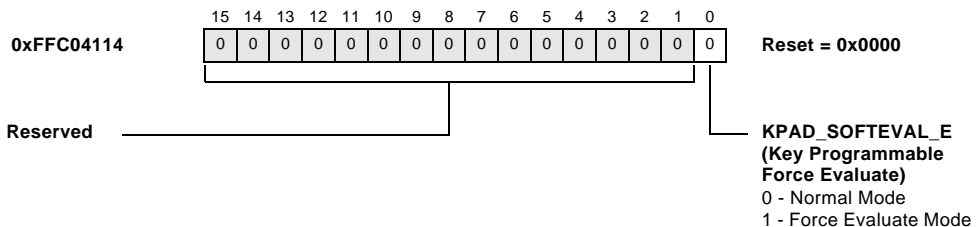


Figure 22-9. Key Software Evaluate Register (KPAD_SOFTEVAL)

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

Programming Examples

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 22-1](#) describes the configuration of the keypad module.

Listing 22-1. Configure Keypad Module

```
/* Configure the prescale register KPAD_PRESCALE */
PO.L = LO(KPAD_PRESCALE);
PO.H = HI(KPAD_PRESCALE);
RO.L = 4; /*0.1 MS WITH 50MHz SCLK */
W[P0] = RO.L;

/* Configure the column drive width and debounce time */
PO.L = LO(KPAD_MSEL);
PO.H = HI(KPAD_MSEL);
RO.L = 0x0909 /* 1 MS WITH 50MHz SCLK, for both column drive
width and debounce time */
W[P0] = RO.L;

/* Configure the KPAD_CTL register to set keypad size to 5 rows
by 6 columns, multiple key press interrupt enabled, keypad module
enabled */
PO.L = LO(KPAD_CTL);
PO.H = HI(KPAD_CTL);
RO.L = 0xB005;
W[P0] = RO.L;

SSYNC;
```

23 SECURE DIGITAL HOST

This chapter describes the ADSP-BF54x Blackfin processor's secure digital host (SDH) interface and includes the following sections:

- [“Overview” on page 23-1](#)
- [“Interface Overview” on page 23-2](#)
- [“Description of Operation” on page 23-3](#)
- [“Functional Description” on page 23-4](#)
- [“Programming Model” on page 23-19](#)
- [“SDH Registers” on page 23-19](#)
- [“Programming Examples” on page 23-39](#)

Overview

The ADSP-BF54x 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.

Interface Overview

Features of the SDH interface include:

- Support for a single MMC, SD Card or SDIO
- Support for 1-bit and 4-bit SD modes (SPI mode is not supported)
- A six-pin external interface with clock, command, and up to 4 data lines
- Card detection using one of the data pins
- Card interface clock generation from SCLK
- SDIO interrupt and read wait features

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 contains 32-bit memory-mapped registers, converts 16-bit PAB accesses to 32-bit register accesses, and generates interrupt requests to the processor core and system. [Figure 23-1](#) shows a block diagram of the SDH interface.

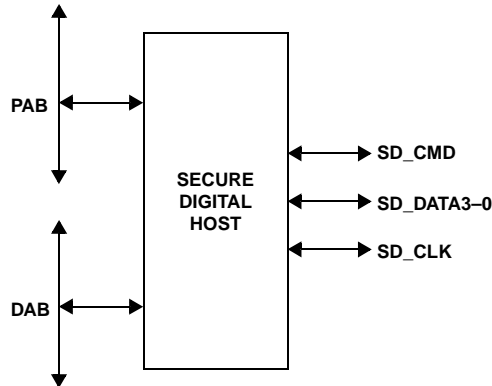


Figure 23-1. SDH Interface Block Diagram

Description of Operation

Table 23-1 lists the six pins in the SDH and their uses in each supported mode.

Table 23-1. SDH Interface

Signal Name	MMC	SD (1-bit)	SD (4-bit)	SDIO (1-bit)	SDIO (4-bit)	Default	Direction
SD_DATA3	Not Used	Not Used	Data bit 3/ Card Detect	Not Used	Data bit 3	Input	Bi-directional
SD_DATA2	Not Used	Not Used	Data bit 2	Read Wait	Data bit 2 or Read Wait	Input	Bi-directional
SD_DATA1	Not Used	Not Used	Data bit 1	Interrupt	Data bit 1 or Interrupt	Input	Bi-directional
SD_DATA0	Data	Data	Data bit 0	Data	Data bit 0	Input	Bi-directional

Functional Description

Table 23-1. SDH Interface (Cont'd)

Signal Name	MMC	SD (1-bit)	SD (4-bit)	SDIO (1-bit)	SDIO (4-bit)	Default	Direction
SD_CMD	Command/Response	Command/Response	Command/Response	Command	Command	Input	Bi-directional
SD_CLK	Clock	Clock	Clock	Clock	Clock	Input	Bi-directional

Functional Description

The following sections describe the MMC, SD, and SDIO functionality.

SDH Clocking

The SDH is a fast, synchronous peripheral. Its peripheral bus interface operates at `SCLK` frequency.

Communication between the clock domains is accomplished using synchronizers in the SDH module.

The clock divider ratio is modified by writing to the SDH clock control register (`SDH_CLK_CTL`). It generates the card clock `SD_CLK` from `SCLK` according to the following formula, where `CLKDIV` is the 8-bit division factor from the `SDH_CLK_CTL` register.

$$SD_CLK\ frequency = \frac{SCLK\ frequency}{2 \times (CLKDIV + 1)}$$

SDH Operation

The SDH sends commands to and receives commands from the multimedia cards. [Figure 23-2](#) shows the command path state machine.

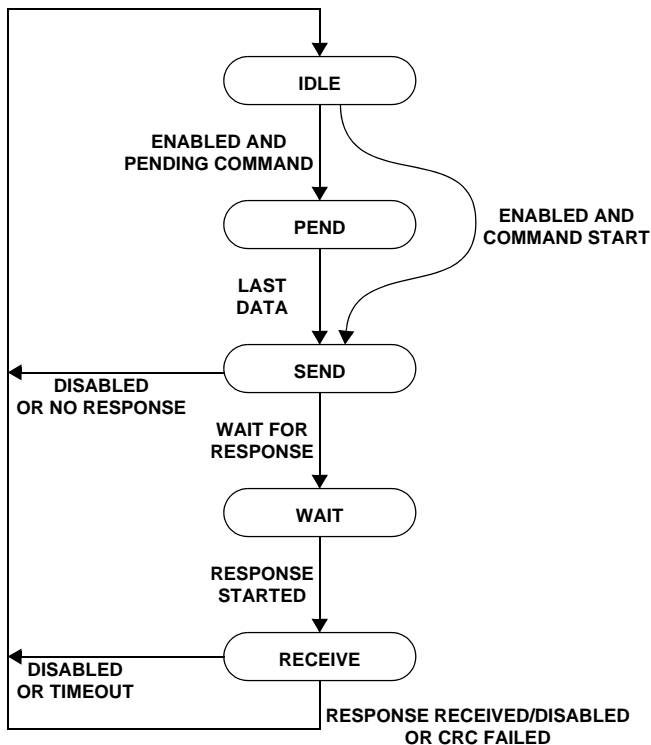


Figure 23-2. Command Path State Machine

When the command register (see “[SDH Command Register \(SDH_COMMAND\)](#)” on page 23-24) is written and the enable bit is set, a command transfer starts.

When the command is sent, the state machine sets the status flags (see “[SDH Status Register \(SDH_STATUS\)](#)” on page 23-30) and enters the IDLE state if a response is not required. If a response is required, the state

Functional Description

machine waits for the response and then, when it is received, a Cyclical Redundancy Code (CRC) is compared to the internally generated code and the appropriate flags are set.

When the WAIT state is entered, a command timer starts running. If the time-out is reached before the state machine moves to the RECEIVE state, the time-out flag is set and the IDLE state is entered. The time-out period has a fixed value of 64 `SD_CLK` clock periods.

If the interrupt bit is set in the command register, the SDH's internal timer is disabled and the state machine waits for an interrupt request from one of the memory cards. If a pending bit is set in the command register, the state machine enters the PEND state, and waits for a signal from the data path sub block that makes the state machine to move to the SEND state. This enables the data counter to trigger the stop command transmission.

Figure 23-3 describes the command transfer.

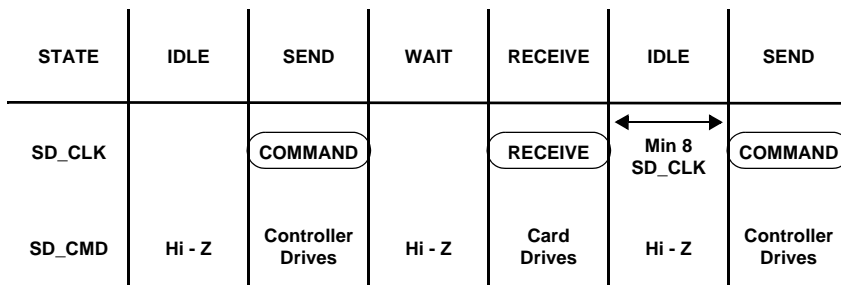


Figure 23-3. SDH Command Transfer



The state machine remains in the IDLE state for at least eight `SD_CLK` periods.

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. Data on SD_CMD is synchronous to the rising edge of SD_CLK. All commands have a fixed length of 48 bits. [Table 23-2](#) shows the command format.

Table 23-2. 48-bit Command Format

Bit Position	Value	Width	Description
47	0	1	Start bit
46	1	1	Transmission bit
[45:40]	–	6	Command index
[39:8]	–	32	Argument
[7:1]	–	7	CRC7
0	1	1	End bit

The SDH supports two response types:

- 48-bit short response (see [Table 23-3](#))
- 136-bit long response (see [Table 23-4](#))

Both response types use CRC error checking. Note that if the response does not contain CRC (CMD1 response), the device driver must ignore the CRC failed status.

Table 23-3. 48-Bit Short Response Format

Bit Position	Value	Width	Description
47	0	1	Start bit
46	0	1	Transmission bit
[45:40]	–	6	Command index
[39:8]	–	32	Argument

Functional Description

Table 23-3. 48-Bit Short Response Format (Cont'd)

Bit Position	Value	Width	Description
[7:1]	–	7	CRC7 (or b#1111111)
0	1	1	End bit

Table 23-4. 136-bit Long Response Format

Bit Position	Value	Width	Description
135	0	1	Start bit
134	1	1	Transmission bit
[133:128]	b#111111	6	Reserved
[127:1]	–	127	CID or CSD (Including internal CRC7)
0	1	1	End bit

The command register contains the command index (six bits sent to the card) and the command type. These determine whether the command requires a response, and whether the response is 48 bits or 136 bits long. The command path implements the status flags of the status register as shown in [Table 23-5](#).

Table 23-5. Command Path Status Flag

Flag	Description
CmdRespEnd	Set if response CRC is successful
CmdCrcFail	Set if response CRC fails
CmdSent	Set when command is sent (if command does not require response)
CmdTimeOut	Response time-out
CmdActive	Command transfer in progress

The command CRC generator calculates the CRC checksum for all bits before the CRC code. This includes the start bit, transmitter bit, command index, and command argument (or card status). The CRC checksum is calculated for the first 120 bits of CID or CSD for the long response format. Note that the start bit, transmitter bit, and the six reserved bits are not used in the CRC calculation. The command CRC checksum is a 7-bit value:

$$CRC[6:0] = \text{Remainder} \frac{x_7 \times M(x)}{G(x)}$$

with:

$$G(x) = x_7 + x_3 + 1$$

and:

$$M(x) = x_{39} \times (\text{start bit}) + \dots + x_0 \times (\text{last bit before CRC})$$

or:

$$M(x) = x_{119} \times (\text{start bit}) + \dots + x_0 \times (\text{last bit before CRC})$$

Functional Description

SDH Data

The SDH data bus width can be programmed using the clock control register (see “[SDH Clock Control Register \(SDH_CLK_CTL\)](#)” on [page 23-23](#)). By setting the WideBus bit in the SDH_CLK_CTL register, bus mode is enabled, and data is transferred at four bits per clock cycle over all four data signals SD_DATA3-0. If the wide bus mode is disabled, only one bit per clock cycle is transferred over SD_DATA0.

The data path state machine operates at SD_CLK frequency. Data on the card bus signals is synchronous to the rising edge of SD_CLK. The state machine has a number of send/receive states as shown in [Figure 23-4](#).

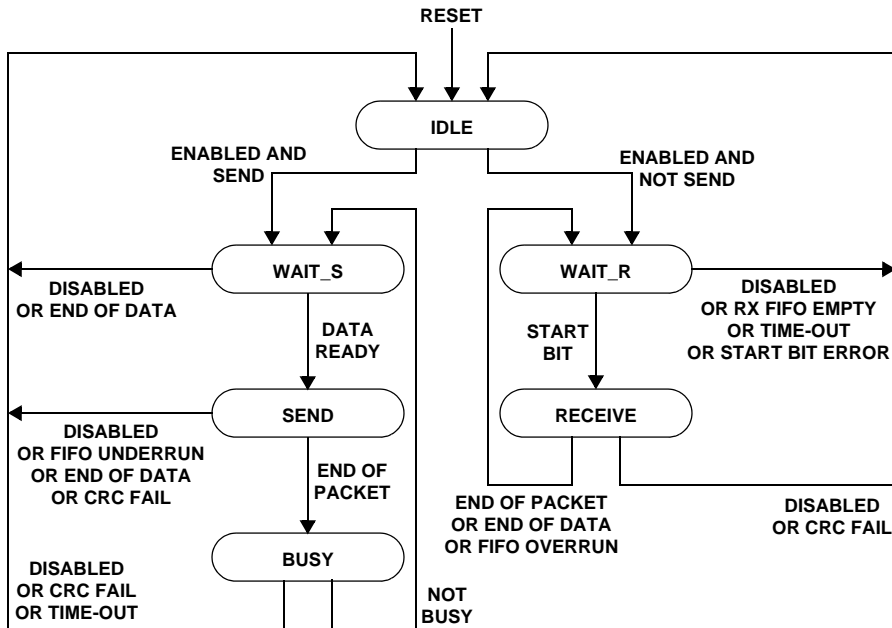


Figure 23-4. SDH Data Path State Machine

WAIT_R

The processor waits for a start bit on `SD_DATA`. The processor starts receiving data if it receives a start bit before a time-out, and loads the data block counter. If it reaches a time-out before it detects a start bit, or a start bit error occurs, the processor moves to the `IDLE` state and sets the time-out status flag.

RECEIVE

Serial data received from a card is packed in bytes and written to the data FIFO. Depending on the transfer mode bit in the data control register, the data transfer mode can be either block or stream.

In block mode, when the data block counter reaches zero, the processor waits until it receives the CRC code. If the received code matches the internally-generated CRC code, the state machine moves to the `WAIT_R` state. If not, the CRC fail status flag is set and the state machine moves to `IDLE` state.

In stream mode, the processor receives data while the data counter is not zero. When the counter is zero, the remaining data in the shift register is written to the data FIFO, and the state machine moves to the `WAIT_R` state.

If a FIFO overrun error occurs, the state machine sets the FIFO error flag and moves to `WAIT_R` state.

Functional Description

SEND

The processor starts sending data to a card. Depending on the transfer mode bit in the data control register, the data transfer mode can be either block or stream:

- In block mode, when the data block counter reaches zero, the processor sends an internally generated CRC code and end bit, and moves to the BUSY state.
- In stream mode, the processor sends data to a card while the enable bit is high and the data counter is not zero. It then moves to the IDLE state.

If a FIFO underrun error occurs, the state machine sets the FIFO error flag and moves to the IDLE state.

The data timer is enabled when the state machine is in the WAIT_R or BUSY state and generates the data time-out error:

- When transmitting data, the time-out occurs if the state machine stays in the BUSY state for longer than the programmed time-out period.
- When receiving data, the time-out occurs if the end of the data is not true and if the state machine stays in the WAIT_R state for longer than the programmed time-out period.

The data counter has two functions:

- To stop a data transfer when it reaches zero. This is the end of the data condition.
- To start transferring a pending command. This is used to send the stop command for a stream data transfer.

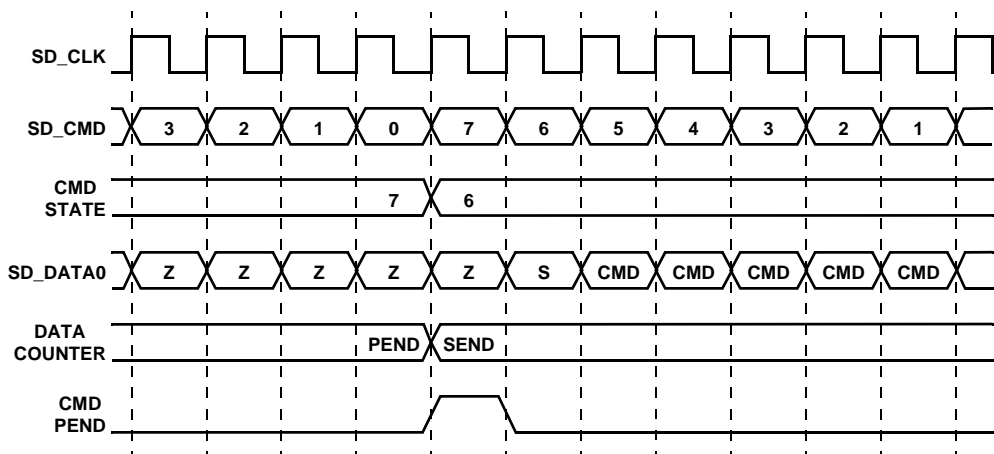


Figure 23-5. Pending Command Start

The SDH module offers several bus modes. In wide bus mode, all four signals `SD_DATA3-0` are used to transfer data, and the CRC code is calculated separately for each data signal. While transmitting data clocks to a card, only `SD_DATA0` is used for CRC token and busy signaling. The start bit must be transmitted on all four data signals at the same time (during the same clock period). If the start bit is not detected on all signals on the same clock edge while receiving data, the state machine sets the start bit error flag and moves to the IDLE state.

Functional Description

The data path also operates in half duplex mode, where data is either sent to a card or received from a card. While no data is being transferred, SD_DATA3-0 are held high by a pull-up in the pads. Data on these signals is synchronous to the rising edge of the clock.

The CRC token status follows each write data block and determines whether a card has received the data block correctly. When the token is received, the card asserts a busy signal by driving SD_DATA0 low.

[Table 23-6](#) shows the CRC token status values.

Table 23-6. CRC Token Status Values

Token	Description
b#010	Card has received error-free data block
b#101	Card has detected a CRC error

[Table 23-7](#) lists the data path status flags.

Table 23-7. SDH Data Path Status Flags

Flag	Description
TxFifoFull	Transmit FIFO is full
TxFifoEmpty	Transmit FIFO is empty
TxFifoHalfEmpty	Transmit FIFO is half empty
TxDataAvlbl	Transmit FIFO data available
TxUnderrun	Transmit FIFO underrun error
RxFifoFull	Receive FIFO is full
RxFifoEmpty	Receive FIFO is empty
RxFifoHalfFull	Receive FIFO is half full
RxDataAvlbl	Receive FIFO data available
RxOverrun	Receive FIFO overrun error
DataBlockEnd	Data block sent/received

Table 23-7. SDH Data Path Status Flags (Cont'd)

Flag	Description
StartBitErr	Start bit not detected on all data signals in wide bus mode
DataCrcFail	Data packet CRC failed
DataEnd	Data end (data counter is zero)
DataTimeout	Data time-out
TxActive	Data transmission in progress
RxActive	Data reception in progress

The data path CRC generator calculates the CRC checksum only for the data bits in a single block, and it is bypassed in data stream mode. The checksum is a 16-bit value:

$$CRC[15:0] = \text{Remainder} \frac{x_{15} \times M(x)}{G(x)}$$

with:

$$G(x) = x_{16} + x_{12} + x_5 + 1$$

where:

$$M(x) = x_{39} \times (\text{start bit}) + \dots + x_0 \times (\text{last bit before CRC})$$

or:

$$M(x) = x_n \times (\text{first data bit}) + \dots + x_0 \times (\text{last data bit})$$

Functional Description

SDH Data FIFO

The data FIFO is a data buffer with transmit and receive logic. The FIFO contains a 32-bit wide, 16-word deep data buffer and transmit and receive logic. Depending on `TxActive` and `RxActive`, the FIFO can be disabled, transmit enabled or receive enabled.

`TxActive` and `RxActive` are mutually exclusive:

- The transmit FIFO refers to the transmit logic and data buffer when `TxActive` is asserted
- The receive FIFO refers to the receive logic and data buffer when `RxActive` is asserted.

Transmit FIFO

If the transmit FIFO is disabled, all status flags are deasserted, and the read and write pointers are reset. The processor asserts `TxActive` when it transmits data. The table below lists the transmit FIFO status flags.

Table 23-8. Transmit FIFO Status Flags

Flag	Description
<code>TxFifoFull</code>	Set to HIGH when all 16 transmit FIFO words contain valid data
<code>TxFifoEmpty</code>	Set to HIGH when the transmit FIFO does not contain valid data
<code>TxFifoHalfEmpty</code>	Set to HIGH when 8 or more transmit FIFO words are empty. This flag is used as DMA request.
<code>TxDataAvbl</code>	Set to HIGH when the transmit FIFO contains valid data. This flag is the inverse of the <code>TxFifoEmpty</code> flag.
<code>TxUnderrun</code>	Set to HIGH when an underrun error occurs. This flag is cleared by writing to the SDH status clear register.

Receive FIFO

If the receive FIFO is disabled, all status flags are deasserted, and the read and write pointers are reset. The processor asserts `RxActive` when it receives data.

Table 23-9. Receive FIFO Status Flags

Flag	Description
<code>RxFifoFull</code>	Set to HIGH when all 16 receive FIFO words contain valid data
<code>RxFifoEmpty</code>	Set to HIGH when the receive FIFO does not contain valid data
<code>RxFifoHalfFull</code>	Set to HIGH when 8 or more receive FIFO words contains valid data. This flag is used as DMA request.
<code>RxDataAvbl</code>	Set to HIGH when the receive FIFO is not empty. This flag is the inverse of the <code>RxFifoEmpty</code> flag.
<code>RxOverrun</code>	Set to HIGH when an overrun error occurs. This flag is cleared by writing to the SDH status clear register.

SDIO Interrupt and Read Wait Support

The SDH accepts interrupt requests from the SDIO card on the `SD_DATA1` pin and generates a read wait interrupt to the card on the `SD_DATA2` pin. There may be multiple interrupt sources within the SDIO card which map to a single interrupt line. Once asserted, the SDIO interrupt remains asserted until the processor determines the source of the interrupt on the SDIO card and clears it. The SDIO interrupt status is indicated in the `SDH_E_STATUS` register (see [“SDH Exception Status Register \(SDH_E_STATUS\)”](#) on page 23-35). The status can be used to generate a processor interrupt. The SDH can also output a read wait interrupt to cause the SDIO to stop a block transfer of data. Once the transfer is stopped, the SDH is able to send commands to the SDIO card. The read wait interrupt is generated by writing to the `SDH_RD_WAIT_EN` register (see [“SDH Read Wait Enable Register \(SDH_RD_WAIT_EN\)”](#) on page 23-38).

Functional Description

In 1-bit SDIO mode, the interrupt and read wait signals use dedicated pins. As a result, the interrupt and read wait requests can occur at any time.

In 4-bit SDIO mode, the interrupt and read wait signals are sent over `SD_DATA` pins. As a result, there are only certain windows of time during which these requests can be asserted. This window of time is referred to as the Interrupt Period. The SDIO interrupt is only sampled and updated in the status register during the Interrupt Period. An interrupt is indicated by a `b#1101` on the `SD_DATA3-0` pins and is terminated by a `b#1111` on the same pins. Likewise, the read wait interrupt is only sent during the Interrupt Period, as indicated by a `b#1011` on the `SD_DATA3-0` pins. If both interrupt and read wait are requested at the same time, `SD_DATA3-0` is `B31001` with `SD_DATA1` driven by the SDIO card and `SD_DATA2` driven by the MMCI.

The timing of the Interrupt Period is mode dependent. In the case of normal single data block transmission, the Interrupt Period is the time between two clocks after the completion of a data pack end bit and the end bit of the next command that will use the DATA lines. In the case of multiple block data transfers, the Interrupt Period is restricted to the two clock period beginning 2 clocks after the end bit of a data block.

MMC/SD Card Detection

The SDH allows software to detect a card when it is inserted into its slot. The `SD_DATA3` pin powers up low due to a special pull-down resistor. When an SD Card is inserted in its slot, the resistance increases and a rising edge is detected by the SDH module. This detection sets the card detect bit in the `SDH_E_STATUS` register and causes an exception to be generated on the SDH interrupt line. Once the card is detected, the SDH pull-down is disabled and the standard pull-up on the pad is enabled. When this pad is used as an alternate pin or a GPIO, the SDH pull-down resistance on the pad can be isolated by programming the pad configura-

tion register. This card detect logic is active even in MMC mode. In this mode, SDH_E_STATUS bit 4 must be cleared to disable interrupts from card detection.

SDH DMA Transfers

The DMA controller can be programmed to transfer data between memory and the SDH's FIFO buffers. This is initiated based on interrupts from the SDH module.

Programming Model

The following sections describe the SDH programming model.

SDH Registers

The 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.

[Table 23-10](#) lists the SDH memory-mapped registers. 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 23-10. SDH Functional Registers

Address Offset	NAME	Type	Access	Description
0xFFC03900	SDH_PWR_CTL	Read/write	Single	SDH power control register on page 23-22
0xFFC03904	SDH_CLK_CTL	Read/write	Single	SDH clock control register on page 23-23

SDH Registers

Table 23-10. SDH Functional Registers (Cont'd)

Address Offset	NAME	Type	Access	Description
0xFFC03908	SDH_ARGUMENT	Read/write	Double	SDH argument register on page 23-24
0xFFC0390C	SDH_COMMAND	Read/write	Single	SDH command register on page 23-24
0xFFC03910	SDH_RESP_CMD	Read only	Single	SDH response command register on page 23-26
0xFFC03914	SDH_RESPONSE0	Read only	Double	SDH response 0 register on page 23-26
0xFFC03918	SDH_RESPONSE1	Read only	Double	SDH response 1 register on page 23-26
0xFFC0391C	SDH_RESPONSE2	Read only	Double	SDH response 2 register on page 23-26
0xFFC03920	SDH_RESPONSE3	Read only	Double	SDH response 3 register on page 23-26
0xFFC03924	SDH_DATA_TIMER	Read/write	Double	SDH data timer register on page 23-26
0xFFC03928	SDH_DATA_LGTH	Read/write	Single	SDH data length register on page 23-27
0xFFC0392C	SDH_DATA_CTL	Read/write	Single	SDH data control register on page 23-27
0xFFC03930	SDH_DATA_CNT	Read only	Single	SDH data counter register on page 23-29
0xFFC03934	SDH_STATUS	Read only	Double	SDH status register on page 23-30
0xFFC03938	SDH_STATUS_CLR	Write only	Single	SDH status clear register on page 23-32
0xFFC0393C	SDH_MASK0	Read/write	Double	SDH interrupt 0 mask register on page 23-33
0xFFC03940	SDH_MASK1	Read/write	Double	SDH interrupt 1 mask register on page 23-33
0xFFC03944	Reserved	–	–	–

Table 23-10. SDH Functional Registers (Cont'd)

Address Offset	NAME	Type	Access	Description
0xFFC03948	SDH_FIFO_CNT	Read only	Single	SDH FIFO counter register on page 23-34
0xFFC0394C ... 0xFFC0397C	Reserved	–	–	–
0xFFC03980	SDH_FIFOx	Read/write	Double	SDH data FIFO registers on page 23-35
0xFFC03984 ... 0xFFC03988	Reserved	–	–	–
0xFFC039c0	SDH_E_STATUS	Read/write	Single	SDH exception status register on page 23-35
0xFFC039C4	SDH_E_MASK	Read/write	Single	SDH exception mask register on page 23-35
0xFFC039C8	SDH_CFG	Read/write	Single	SDH configuration register on page 23-36
0xFFC039CC	SDH_RD_WAIT_EN	Read/write	Single	SDH read wait enable register on page 23-38
0xFFC039D0 ... 0xFFC039EC	SDH_PIDx	Read only	Single	SDH peripheral identification registers (8-bit values) on page 23-38

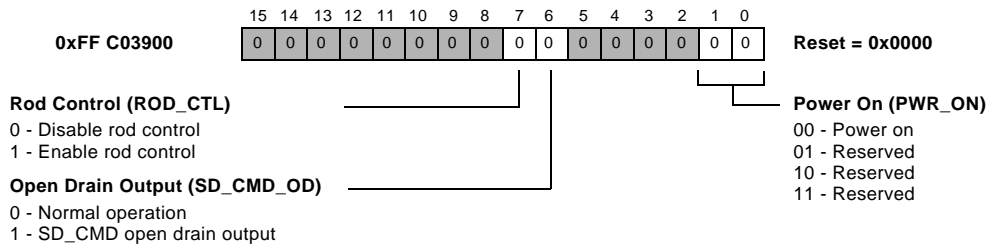
SDH Registers

SDH Power Control Register (SDH_PWR_CTL)

The SDH_PWR_CTL register (see [Figure 23-6](#)) contains bits that control the power consumption by the SDH module. After a data write, data cannot be written to this register for five SCLK cycles.

SDH Power Control Register (SDH_PWR_CTL)

Read/Write



NOTE:
Bits 32–16 (not shown) are reserved

Figure 23-6. SDH Power Control Register (SDH_PWR_CTL)

SDH Clock Control Register (SDH_CLK_CTL)

The SDH_CLK_CTL register (see [Figure 23-6](#)) contains bits that control the SDH clock.

SDH Clock Control Register (SDH_CLK_CTL)

Read/Write

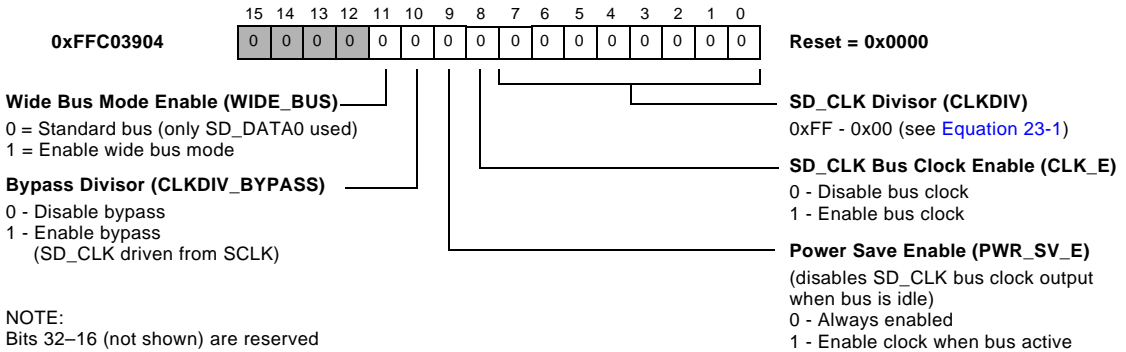


Figure 23-7. SDH Clock Control Register (SDH_CLK_CTL)

While the SDH module is in identification mode, the SD_CLK frequency must be less than 400 kHz. The clock frequency can be changed to the maximum card bus frequency when relative card addresses are assigned to all cards. The CLKDIV bit field value determines the SDH clock frequency as shown in [Equation 23-1](#).

Equation 23-1. SD_CLK Frequency Calculation

$$SD_CLK\ frequency = \frac{SCLK\ frequency}{2 \times (CLKDIV + 1)}$$

After a data write, data cannot be written to the SDH_CLK_CTL register for five SCLK cycles.

SDH Argument Register (SDH_ARGUMENT)

The `SDH_ARGUMENT` register—a read-write register located at offset `0xFFC03908` with reset value `0x0000`—contains a 32-bit command argument, which is sent to a card as part of a command message. If a command contains an argument, it must be loaded into the argument register before writing a command to the command register. For more information on commands and arguments, see “[SDH Command Register \(SDH_COMMAND\)](#)”.

SDH Command Register (SDH_COMMAND)

The `SDH_COMMAND` register (see [Figure 23-8](#)) contains the command index and command type bits. The command index is sent to a card as part of a command message. The command type bits control the command path state machine. Writing a 1 to the enable bit starts the command send operation, while clearing the bit disables the command state machine.

SDH Command Register (SDH_COMMAND)

Read/Write

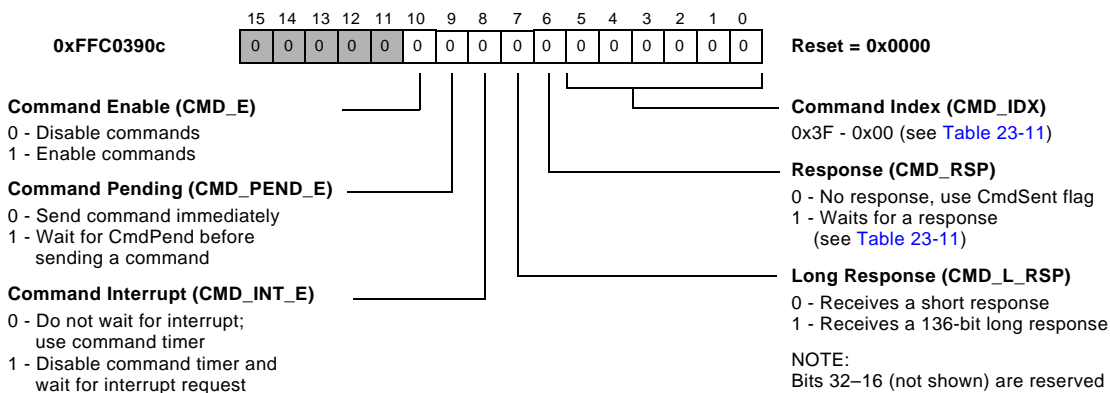


Figure 23-8. SDH Command Register (SDH_COMMAND)

After a data write, data cannot be written to the SDH_COMMAND register for five SCLK cycles.

The SDH command argument values are loaded into the SDH_ARGUMENT register before loading the command value in the SDH_COMMAND register.

Table 23-11 lists the SDH command response values.

Table 23-11. SDH Command Response Type Values

Response	LongRsp	Description
0	0	No response, expect CmdSent flag
0	1	No response, expect CmdSent flag
1	0	Short response, expect CmdRespEnd or CmdCrcFail flag
1	1	Long response, expect CmdRespEnd or CmdCrcFail flag

SDH Response Command Register (SDH_RESP_CMD)

The SDH_RESP_CMD register (see Figure 23-8) contains the command index field of the last command response received. If the command response transmission does not contain the command index field (long response), the RESP_CMD field is unknown, although it must contain b#111111 (the value of the reserved field from the response).

SDH Command Response Register (SDH_RESP_CMD)

Read only

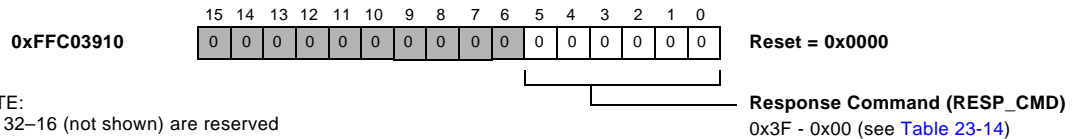


Figure 23-9. SDH Command Response Register (SDH_RESP_CMD)

SDH Response Registers (SDH_RESPONSEx)

The SDH_RESPONSE3-0 registers—read-only registers located at offsets 0xFFC03914 through 0xFFC03920 with reset value 0x0000—contain 32-bit values for the read-only status of the card, which is part of the received response. (See Table 23-12.) The card status size can be 32 or 127 bits, depending on the response type. The most significant bit of the card status is received first. The SDH_RESPONSE3 register least significant bit is always zero.

Table 23-12. SDH_RESPONSE3-0 Response Register Types

Description	Short response	Long response
SDH_RESPONSE0	Card status [31:0]	Card status [127:96]
SDH_RESPONSE1	Unused	Card status [95:64]
SDH_RESPONSE2	Unused	Card status [63:32]
SDH_RESPONSE3	Unused	Card status [31:0]

SDH Data Timer Register (SDH_DATA_TIMER)

The SDH_DATA_TIMER register—a read-write register located at offset 0xFFC03924 with reset value 0x0000—contains a 32-bit value for the data timeout period, in card bus clock periods. A counter loads the value from the data timer register, and starts to decrement when the data path state machine enters the WAIT_R or BUSY state. If the timer reaches zero while the state machine is in either of these states, the timeout flag is set. A data transfer must be written to the data timer register and the data length register before being written to the data control register.

SDH Data Length Register (SDH_DATA_LGTH)

The SDH_DATA_LGTH register (see [Figure 23-10](#)) contains a 16-bit value for the number of data bytes to be transferred. The value is loaded into the data counter when the transfer starts.

SDH Data Length Register (SDH_DATA_LGTH)

Read/Write

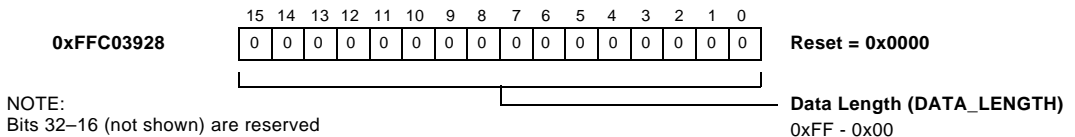


Figure 23-10. SDH Data Length Register (SDH_DATA_LGTH)

SDH Data Control Register (SDH_DATA_CTL)

The SDH_DATA_CTL register (see [Figure 23-11](#)) controls the data path state machine. After a data write, data cannot be written to this register for five SCLK cycles.

SDH Data Control Register (SDH_DATA_CTL)

Read/Write

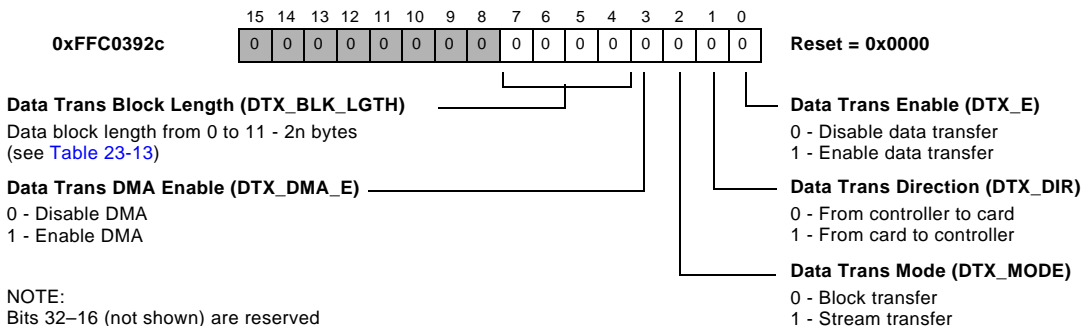


Figure 23-11. SDH Data Control Register (SDH_DATA_CTL)

SDH Registers

Data transfer starts if a 1 is written to the enable bit. Depending on the direction bit, the data path state machine moves to the WAIT_S or the WAIT_R state. There is no need to clear the enable bit after data transfer. [Table 23-13](#) describes the data block length if block data transfer mode is selected.

Table 23-13. Data Block Length

Block size	Block length
0	20= 1 byte card status [127:96]
1	21 = 2 bytes
...	–
11	211= 2048 bytes
12–15	reserved

After a data write, data cannot be written to the SDH_DATA_CTL register for five SCLK cycles.

SDH Data Counter Register (SDH_DATA_CNT)

The SDH_DATA_CNT register (see [Figure 23-12](#)) loads the 16-bit value from the data length register when the data path state machine moves from the IDLE state to the WAIT_S or WAIT_R state. As data is transferred, the counter decrements the value until it reaches 0. The state machine then moves to IDLE state and the data status end flag is set.

SDH Data Counter Register (SDH_DATA_CNT)

Read only

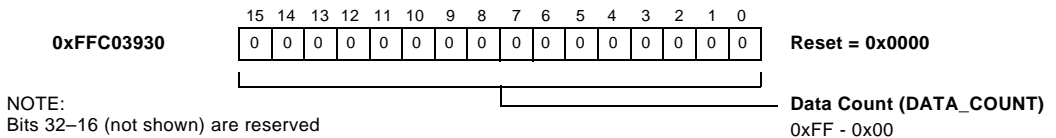


Figure 23-12. SDH Data Counter Register (SDH_DATA_CNT)

Only read the SDH_DATA_CNT register when the data transfer is complete.

SDH Status Register (SDH_STATUS)

The SDH_STATUS register (see [Figure 23-13](#) and [Figure 23-14](#)) is read-only. It contains two types of flags: static and dynamic.

SDH Status Register (SDH_STATUS), Bits [15:0]

Read Only

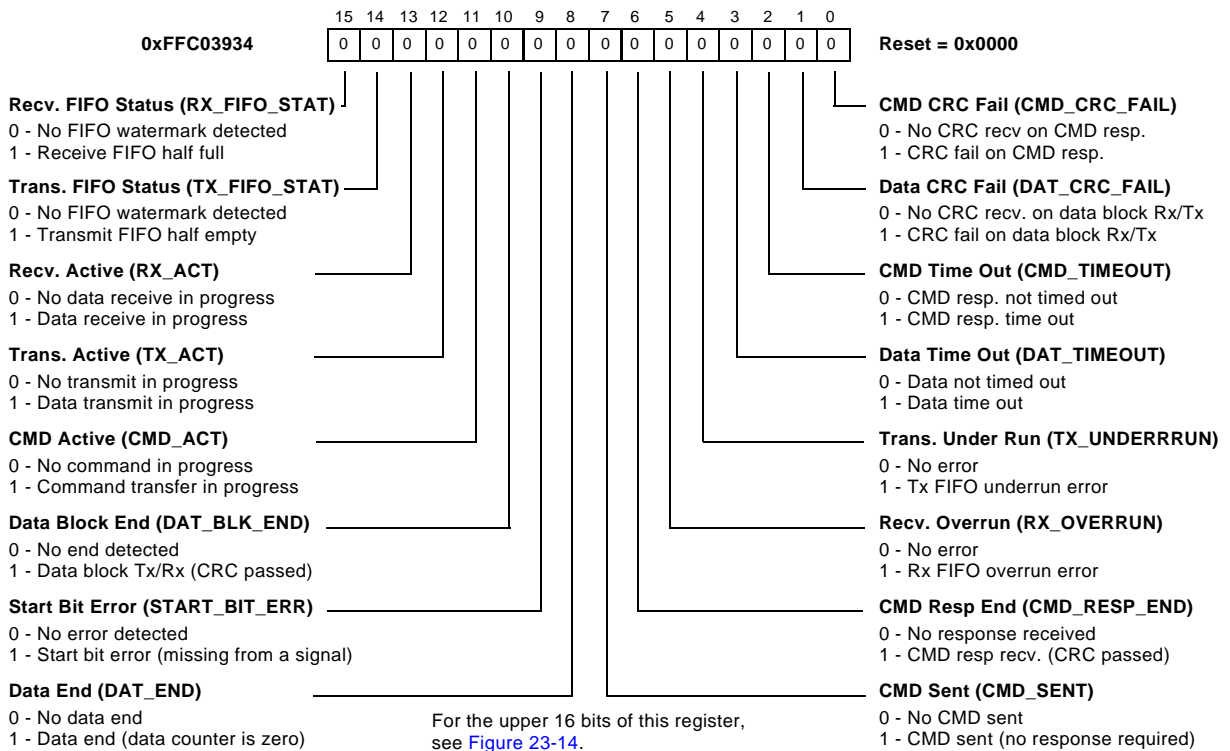


Figure 23-13. SDH Status (SDH_STATUS) Register, Bits [15:0]

The static flags (bits [10:0]) remain asserted until they have been cleared by writing to the Clear register. the dynamic flags (bits [21:11]) change state depending on the state of the underlying logic. For example, FIFO full and empty flags are asserted and deasserted as data is written to or read from the FIFO.

SDH Status Register (SDH_STATUS). Bits [31:16]

Read Only

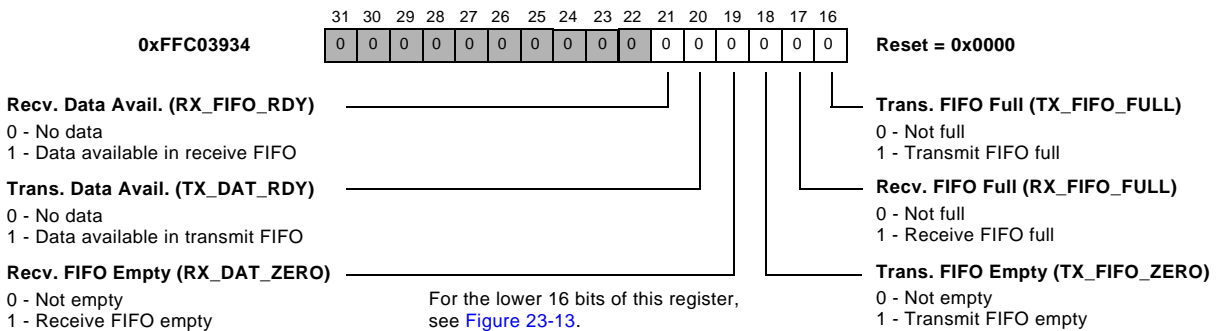


Figure 23-14. SDH Status Register (SDH_STATUS), Bits [31:16]

SDH Status Clear Register (SDH_STATUS_CLR)

The SDH_STATUS_CLR register (see [Figure 23-15](#)) is write-only. The corresponding static status flag can be cleared by writing a 1 to the corresponding bit in the register.

SDH Status Clear Register (SDH_STATUS_CLR)

Write Only (W1C)

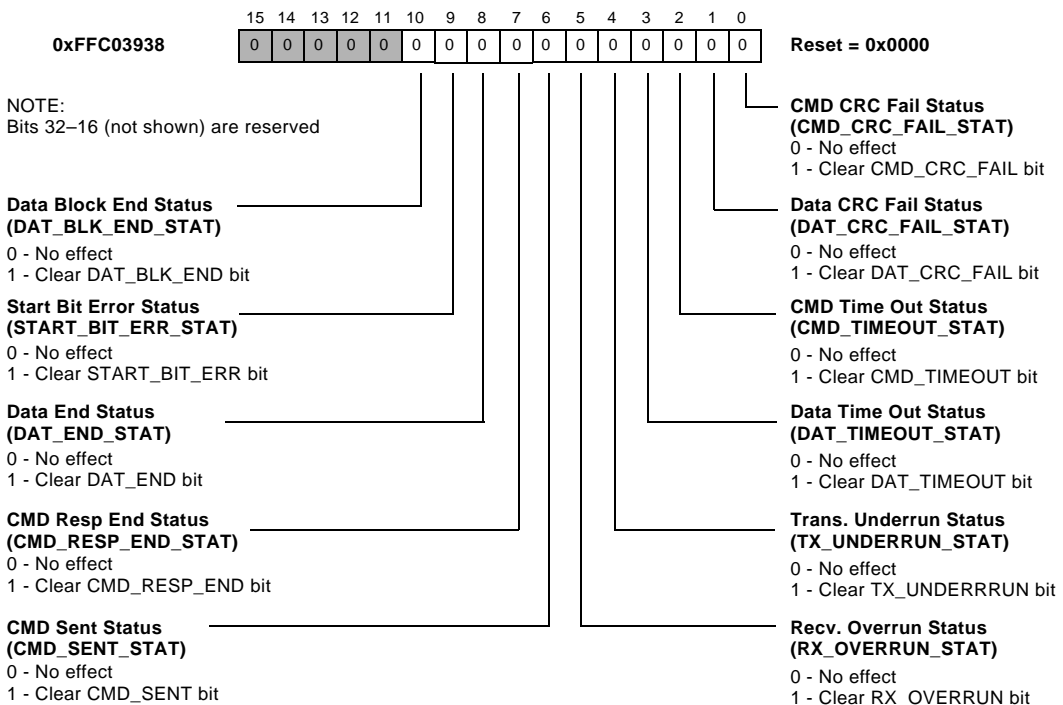


Figure 23-15. SDH Status Clear Register (SDH_STATUS_CLR)

SDH Interrupt Mask Registers (SDH_MASKx)

The SDH_MASK_x registers (see [Figure 23-16](#) and [Figure 23-17](#)) are read-write. The bits in the SDH_MASK₀ register determine which status flags in the SDH_STATUS register generate an interrupt request. To enable an interrupt for a status flag, set the corresponding mask bit to 1.

SDH Interrupt Mask Registers (SDH_MASKx), Bits [15:0]

Read/Write

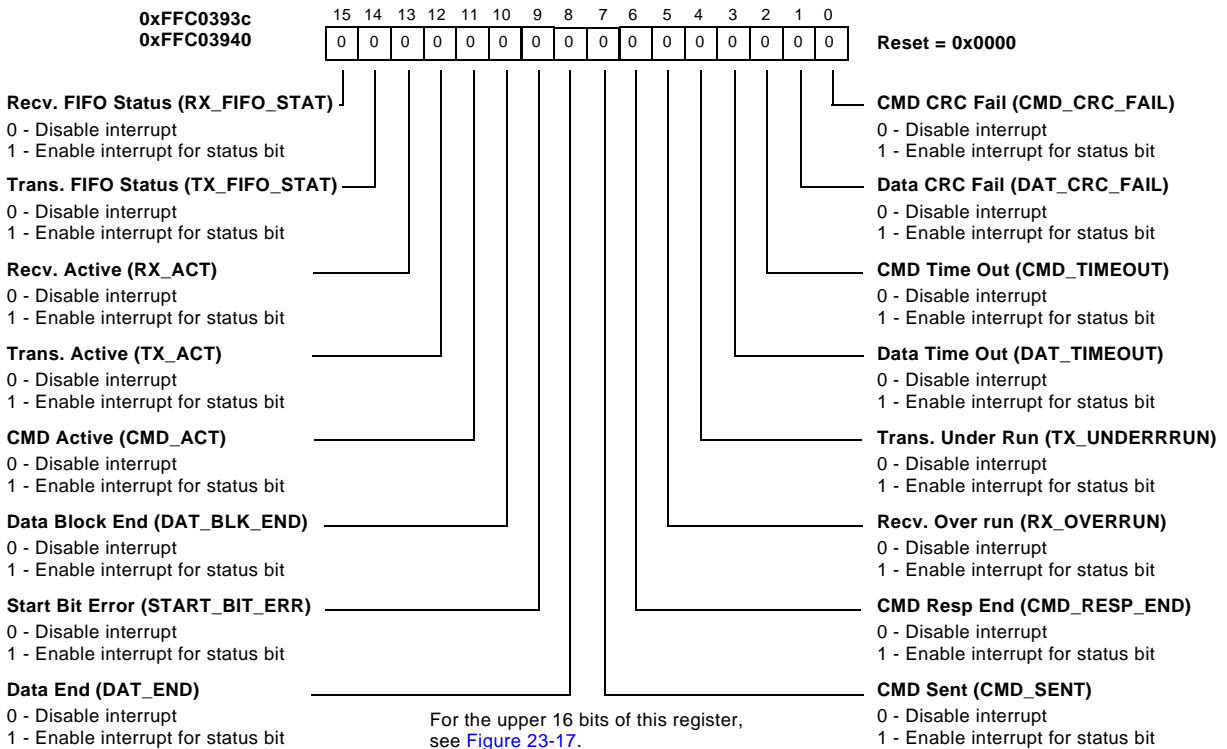


Figure 23-16. SDH Interrupt Mask Registers (SDH_MASK_x), Bits [15:0]

SDH Registers

SDH Interrupt Mask Registers (SDH_MASKx), Bits [31:16]

Read/Write

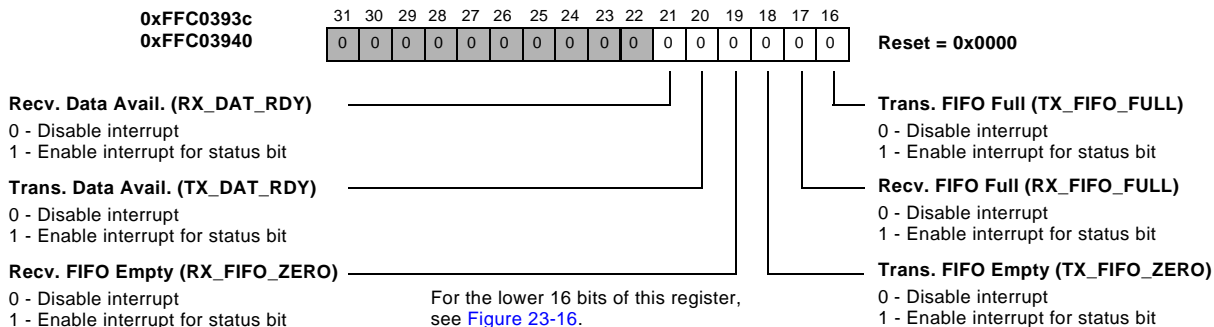


Figure 23-17. SDH Interrupt Mask Registers (SDH_MASKx), Bits [31:16]

SDH FIFO Counter Register (SDH_FIFO_CNT)

The SDH_FIFO_CNT register contains a value indicating the remaining number of words to be written to or read from the FIFO. The FIFO counter loads the value from the SDH_DATA_LGTH register when the DTX_E enable bit is set in the SDH_DATA_CTL register. If the data length is not word-aligned (multiple of 4), the remaining 1 to 3 bytes are regarded as word.

SDH FIFO Counter Register (SDH_FIFO_CNT)

Read only

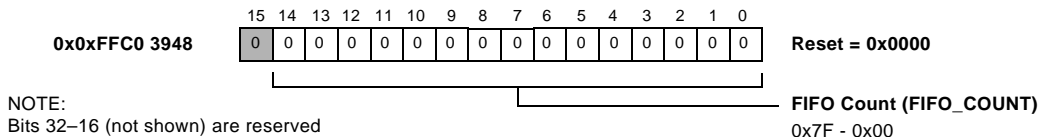


Figure 23-18. SDH FIFO Counter Register (SDH_FIFO_CNT)

SDH Data FIFO (SDH_FIFOx) Registers

The SDH_FIFOx receive and transmit FIFO registers—read/write registers located starting at offset 0xFFC03980 with reset value 0x0000—can be read or written as 32-bit registers. The FIFOs contain 16 entries on 16 sequential addresses. This allows the processor to read from and write to the FIFOs.

SDH Exception Status Register (SDH_E_STATUS)

The SDH_E_STATUS register (see [Figure 23-19](#)) contains bits that indicate exceptions, SD card detection, and FIFO accesses. These bits can be used to generate interrupts to the processor if unmasked in the SDH_E_MASK register.

SDH Exception Status Register (SDH_E_STATUS)

Read only/W1C

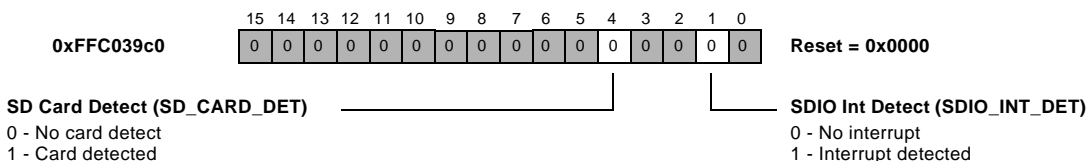


Figure 23-19. SDH Exception Status Register (SDH_E_STATUS)

SDH Exception Mask Register (SDH_E_MASK)

The SDH_E_MASK register (see [Figure 23-20](#)) contains mask bits for SDH error and status signals. When the mask bits are set, these signals generate interrupts. All error conditions are combined into a single interrupt line

SDH Registers

that is routed to the processor core and system interrupt controllers. The `RXFIFOEMPTY` and `TXFIFOFULL` interrupts can trigger the processor core or DMA to move data in or out of the SDH FIFOs.

SDH Exception Mask Register (SDH_E_MASK)

Read/Write

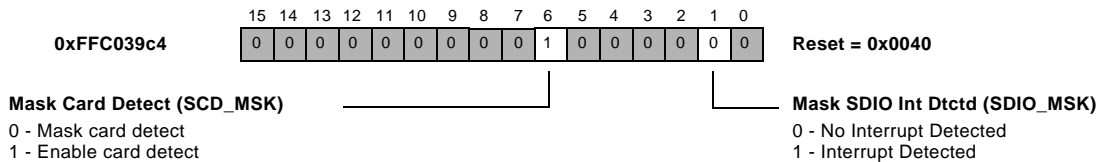


Figure 23-20. SDH Exception Mask Register (SDH_E_MASK)

SDH Configuration Register (SDH_CFG)

The `SDH_CFG` register (see [Figure 23-21](#)) contains bits that enable and disable portions of the SDH module. The clock enable bit must be set to enable the SDH for operation. In addition, the underflow and overflow errors as well as 4-bit SDIO mode may be individually enabled or disabled. The `MWE` bit can be set to allow SDIO interrupts outside the specified one cycle window. The `SDMMC` reset bit is included to reset the SDH module for debug purposes. This bit is a Write-1-to-Action (W1A) bit, and always reads as 0. The `PUP_SDDAT`, `PUP_SDDAT3`, and `PD_SDDAT3` bits enable pull-up and pull-down resistors for the `SD_DATAx` and `SD_CMD` pins.

SDH Configuration Register (SDH_CFG)

Read/Write, Write-1-Act

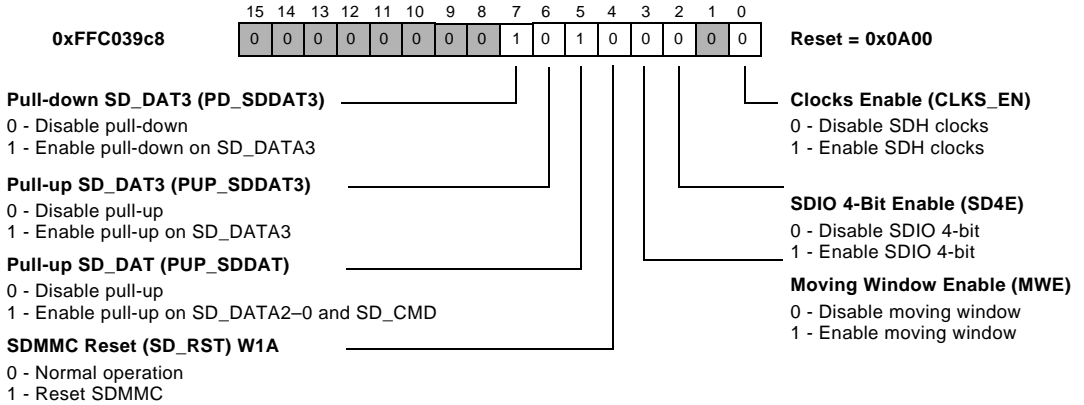


Figure 23-21. SDH Configuration Register (SDH_CFG)



Do not set PD_SDDAT3 and PUP_SDDAT3 at the same time.

SDH Registers

SDH Read Wait Enable Register (SDH_RD_WAIT_EN)

The SDH_RD_WAIT_EN register (see [Figure 23-22](#)) contains one bit that, when set, issues a read wait request to an SDIO card. Once software is ready to resume data transfer, this bit must be cleared. This functionality applies to both 1-bit and 4-bit SDIO cards.

SDH Read Wait Enable Register (SDH_RD_WAIT_EN)

Read/Write

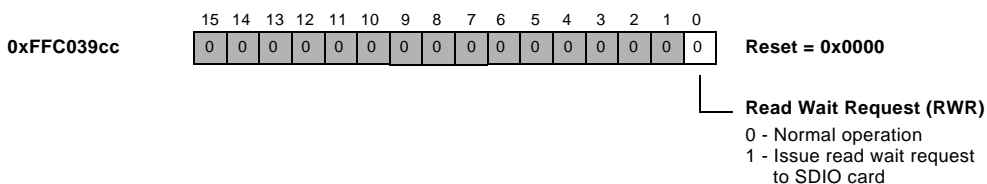


Figure 23-22. SDH Read Wait Enable Register (SDH_RD_WAIT_EN)

SDH Identification Registers (SDH_PIDx)

The SDH_PIDx registers (see [Table 23-14](#)) contain a fixed value at reset used to identify the peripheral, its configuration, and revision. There are a total of eight 8-bit identification registers.

Table 23-14. SDH ID Registers (SDH_PIDx)

Address	Name	Width	Value	Type	Access	Description
0xFFC039D0	SDH_PID0	8	0x80	Read only	Double	SDH Peripheral Identification
0xFFC039D4	SDH_PID1	8	0x11	Read only	Double	SDH Peripheral Identification
0xFFC039D8	SDH_PID2	8	0x04	Read only	Double	SDH Peripheral Identification
0xFFC039DC	SDH_PID3	8	0x00	Read only	Double	SDH Peripheral Identification
0xFFC039E0	SDH_PID4	8	0x0D	Read only	Double	SDH Peripheral Identification
0xFFC039E4	SDH_PID5	8	0xF0	Read only	Double	SDH Peripheral Identification

Table 23-14. SDH ID Registers (SDH_PIDx) (Cont'd)

Address	Name	Width	Value	Type	Access	Description
0xFFC039E8	SDH_PID6	8	0x05	Read only	Double	SDH Peripheral Identification
0xFFC039EC	SDH_PID7	8	0xB1	Read only	Double	SDH Peripheral Identification

The SDH_PIDx registers are read-only. The values of the bits can be grouped into two 32-bit words—the first word comprehends SDH_PID[3:0] of value 0x00041180, and the second word comprehends SDH_PID[7:4] of value 0xB105F00D.

Programming Examples

The following programming examples describe the programmable features of the ADSP-BF54x processor's secure digital I/O.

Listing 23-1. Secure Digital I/O Programming Example

```
/* TBD ... TBD ... TBD ... TBD */
```

Programming Examples

24 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 24-1](#)
- [“Description of Operation” on page 24-4](#)
- [“Functional Description” on page 24-19](#)
- [“Programming Model” on page 24-43](#)
- [“ATAPI Registers” on page 24-48](#)
- [“ATAPI Standards Reference” on page 24-74](#)

Interface Overview

The ATAPI interface supports all ATA hardware protocol transfers and the complete set of 80 ATAPI commands.

Interface Overview

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 24-1 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

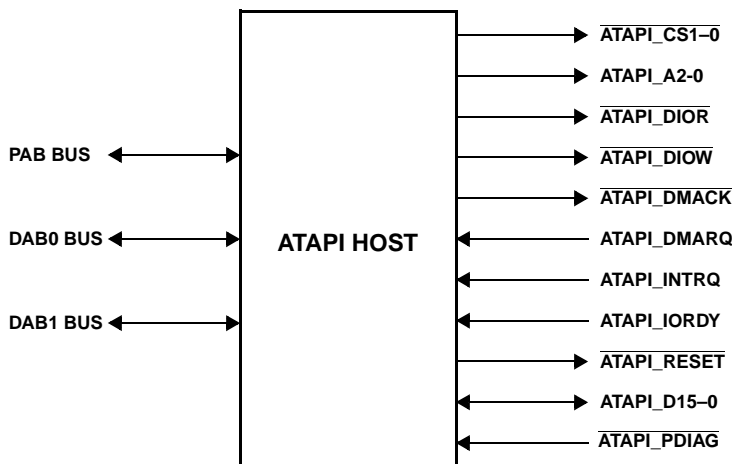


Figure 24-1. ATAPI Block Diagram

The ATAPI shares its pins with other peripherals on chip. Please refer to the “General-Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)* for more information.

Table 24-1 lists the signal pins for the ATAPI block.

Table 24-1. ATAPI Signals Summary

Signal	Dir	Description
$\overline{\text{ATAPI_CS1-0}}$	O	Chip select signals from the host used to select the command block or control block registers. When $\overline{\text{DMACK}}$ is asserted, $\overline{\text{ATAPI_CS1-0}}$ 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.
$\overline{\text{ATAPI_DIOR}}$ $\overline{\text{ATAPI_HDMARDY}}$ ATAPI_HSTROBE	O	$\overline{\text{ATAPI_DIOR}}$ is the strobe signal asserted by the host to read the device register or data port.
$\overline{\text{ATAPI_DIO\overline{W}}}$ ATAPI_STOP	O	$\overline{\text{ATAPI_DIO\overline{W}}}$ is the strobe signal asserted by the host to write the device register or data port
$\overline{\text{ATAPI_DMACK}}$	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 $\overline{\text{ATAPI_DMACK}}$. The host can pause the DMA transfer by deasserting ATAPI_DMARQ. At the same time, $\overline{\text{ATAPI_DMACK}}$ 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.
$\overline{\text{ATAPI_IORDY}}$ $\overline{\text{ATAPI_DDMARDY}}$ ATAPI_DSTROBE	I	The device can create wait state when it is not ready to respond for any host register access (read or write).
$\overline{\text{ATAPI_RESET}}$	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
$\overline{\text{ATAPI_PDIAG}}$	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.

Description of Operation

- 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 24-2](#)).

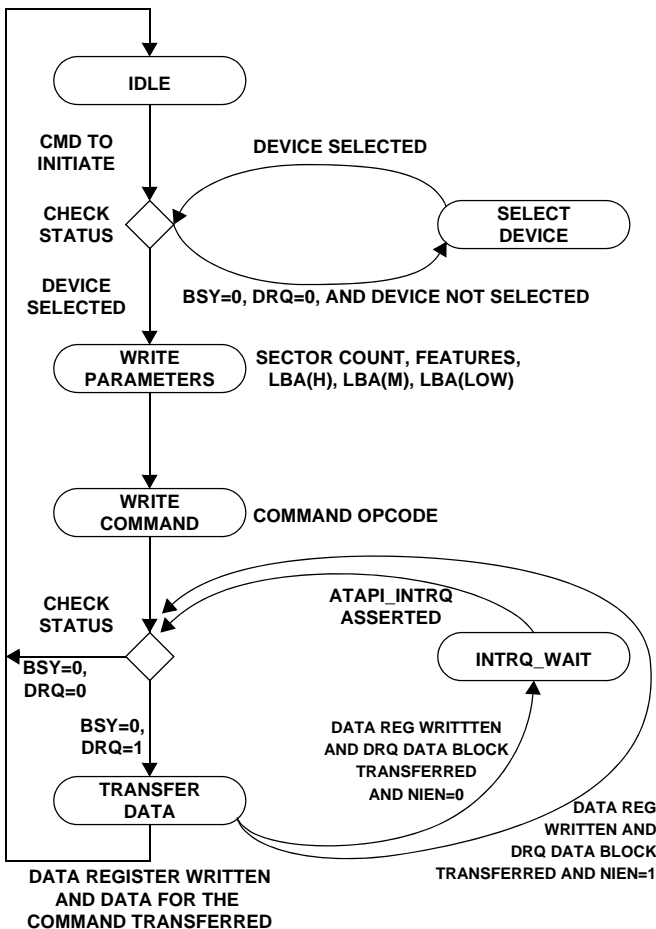


Figure 24-2. PIO Data-Out Protocol State Machine (Device Write)

Description of Operation

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 set 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.
- 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 24-3](#)).

Description of Operation

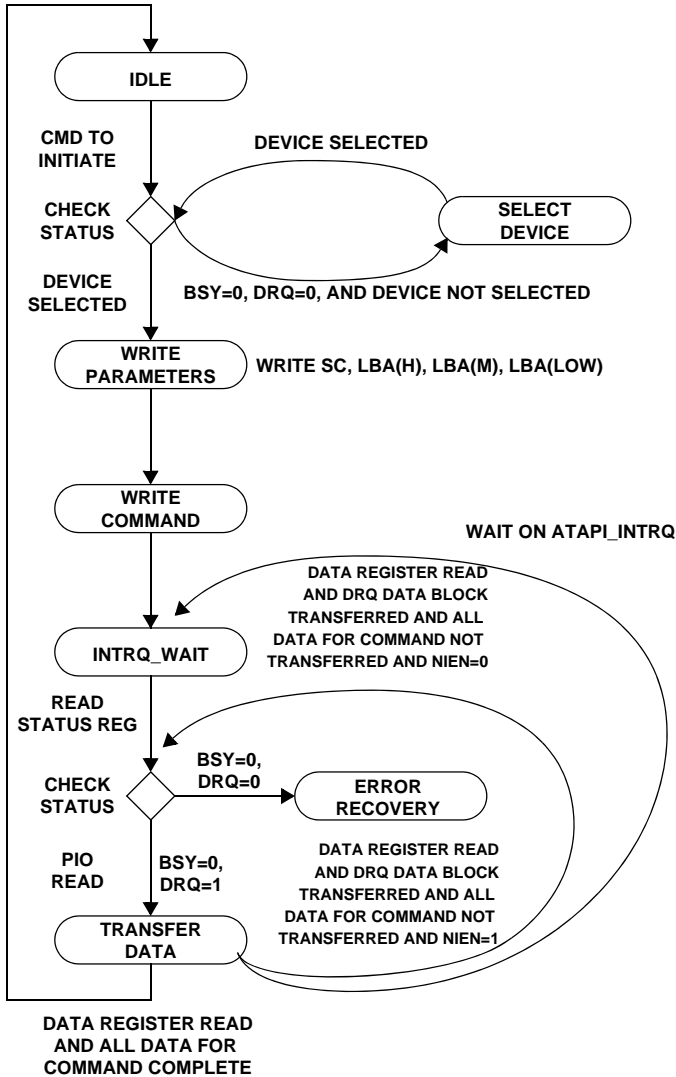


Figure 24-3. PIO Data-In State Machine (Device Read)

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.
- 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.

Description of Operation

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.

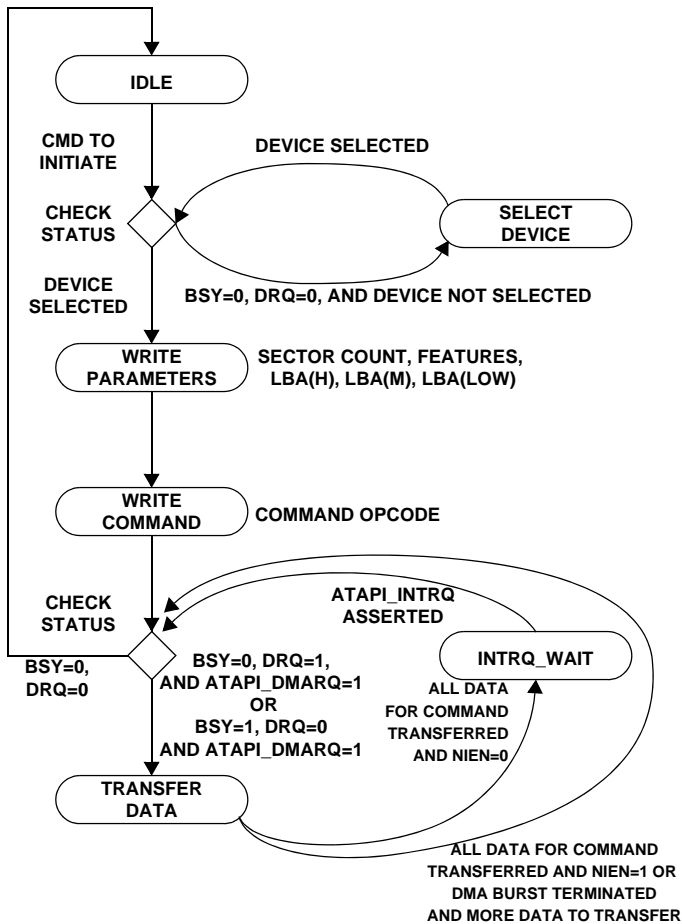


Figure 24-4. Host DMA State Machine

Description of Operation

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 $\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\#}}$ 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 $\overline{\text{ATAPI_DIO\#}}/\overline{\text{ATAPI_DIOR}}$ 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 $\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\#}}$ 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 $\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\#}}$ 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 `ATAPI_DIOW/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 `ATAPI_DIOR/ATAPI_DIOW` pulse. The last word for the burst is then transferred by the negation of the current `ATAPI_DIOR` or `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.

Description of Operation

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 IDENTIFY 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 $\overline{\text{ATAPI_HDMARDY}}$. The device stops generating the ATAPI_DSTROBE edges with in t_{RES} -(75ns mode0 to 50ns mode2) of the host negating $\overline{\text{ATAPI_HDMARDY}}$. 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_{RES} 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 $\overline{\text{ATAPI_HDMARDY}}$ 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 $\overline{\text{ATAPI_HDMARDY}}$.

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.

Description of Operation

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 `ATAPI_DMACK` asserted and the `HSTOP` de-asserted. This should not make the device start a termination sequence, as the `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 `ATAPI_DMACK`.

Device Pausing the Ultra DMA Data-Out Transfer

The device can pause an Ultra DMA data-out burst by negating `ATAPI_DDMARDY`. The ATAPI host enters into a pause state and waits until the device asserts `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 `tRP`. The ATAPI host enters the pause state once it sees

the $\overline{\text{ATAPI_DDMARDY}}$ 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 $\text{ULTRA_OUT_TERMINATED}$ 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, please refer to the ATA/ATAPI-6 Specification and *ADSP-BF54x Blackfin 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 $\overline{\text{ATAPI_RESET}}$ pin of the device. When the $\overline{\text{ATAPI_RESET}}$ signal is asserted, the connected devices execute the hardware reset protocol. The host should respond as described below:

1. Assert $\overline{\text{ATAPI_RESET}}$ 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 $\overline{\text{ATAPI_RESET}}$ 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.

Functional Description

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).

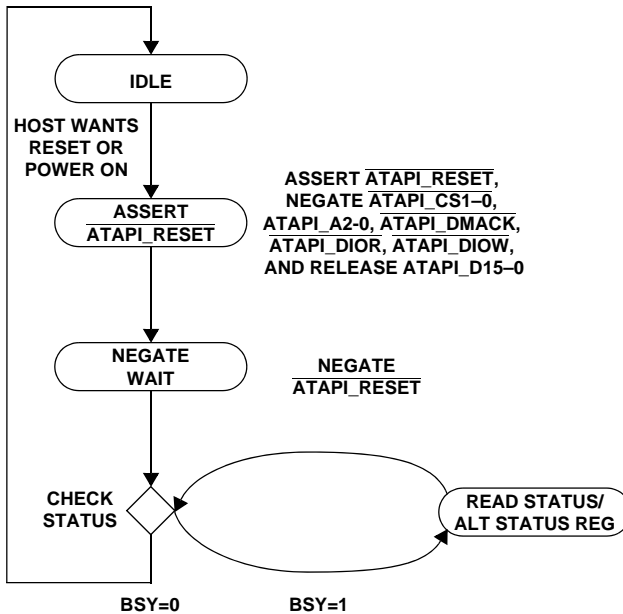


Figure 24-5. Power-On and Hardware Reset Protocol

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 24-6 below describes the protocol for device selection.

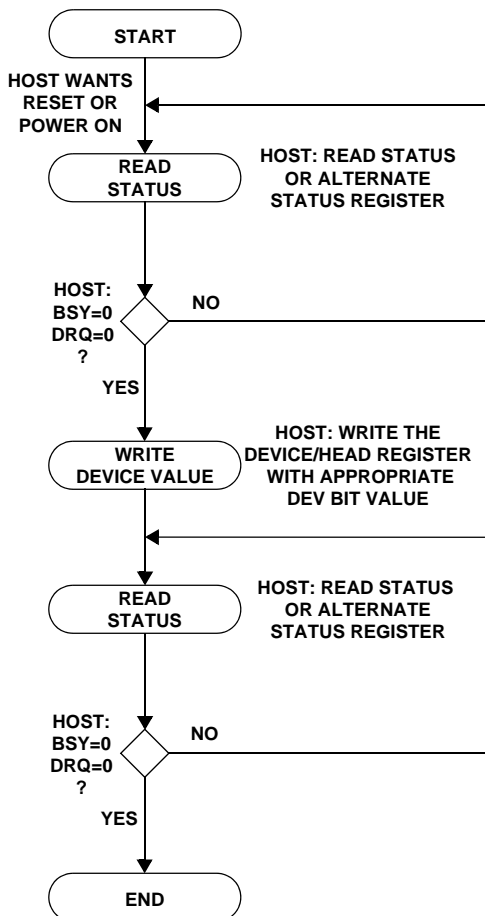


Figure 24-6. Device Selection Protocol

Functional Description

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 24-7](#). The following notes apply to [Figure 24-7](#).

- 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 .

- $\overline{\text{ATAPI_DMACK}}$ is negated during a PIO data transfer.

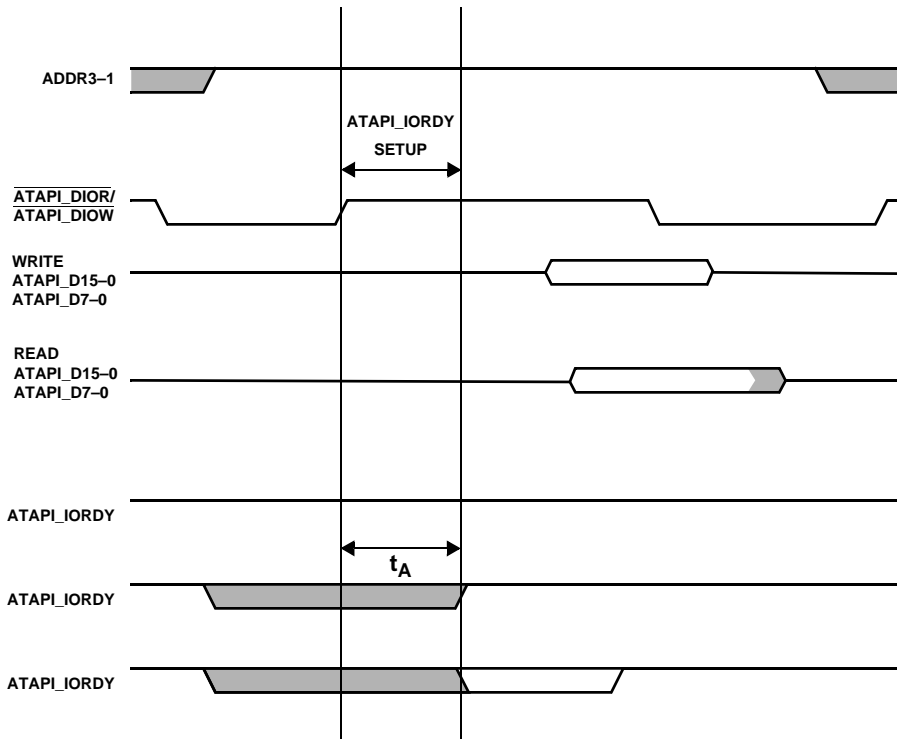


Figure 24-7. PIO Data Transfer to/from the Device Timing Diagram

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

Functional Description

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 24-8. 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.

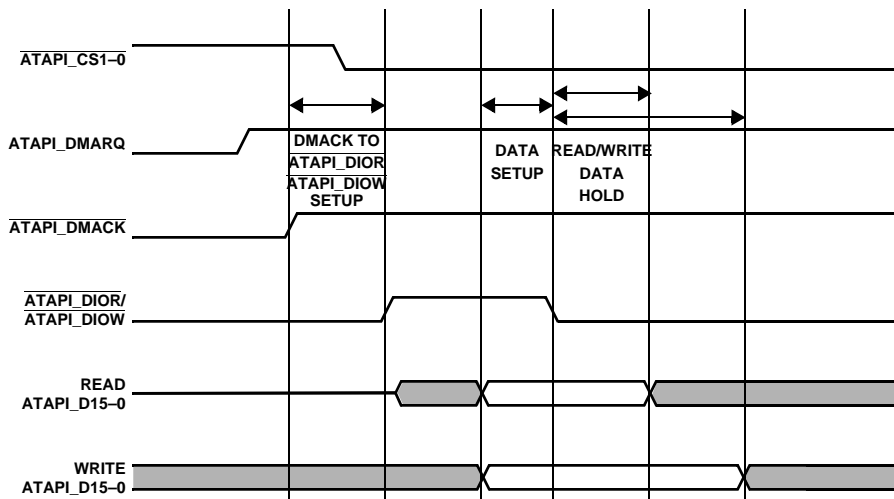


Figure 24-8. Initiating a Multiword DMA Burst

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 24-9](#)).

Table 24-2. Multiword DMA Transfer Timing Table

Multiword DMA Timing Parameters	
t_0	Cycle time ¹
t_D	$\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\overline{W}}}$ asserted pulse width ¹
t_E	$\overline{\text{ATAPI_DIOR}}$ data access
t_F	$\overline{\text{ATAPI_DIOR}}$ data hold
t_G	$\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\overline{W}}}$ data setup
t_H	$\overline{\text{ATAPI_DIO\overline{W}}}$ data hold
t_I	$\overline{\text{ATAPI_DMACK}}$ to $\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\overline{W}}}$ setup
t_J	$\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\overline{W}}}$ to $\overline{\text{ATAPI_DMACK}}$ hold
t_{KR}	$\overline{\text{ATAPI_DIOR}}$ negated pulse width ¹
t_{KW}	$\overline{\text{ATAPI_DIO\overline{W}}}$ negated pulse width ¹
t_{LR}	$\overline{\text{ATAPI_DIOR}}$ to $\overline{\text{ATAPI_DMARQ}}$ delay
t_{LW}	$\overline{\text{ATAPI_DIO\overline{W}}}$ to $\overline{\text{ATAPI_DMARQ}}$ delay
t_M	$\overline{\text{ATAPI_CS1-0}}$ valid to $\overline{\text{ATAPI_DIOR}}/\overline{\text{ATAPI_DIO\overline{W}}}$
t_N	$\overline{\text{ATAPI_CS1-0}}$ hold
t_Z	$\overline{\text{ATAPI_DMACK}}$ to read data released

1. For exact timing information, please refer to the ATA/ATAPI-6 Specification and BF54x Blackfin Embedded Processor data sheet.

Functional Description

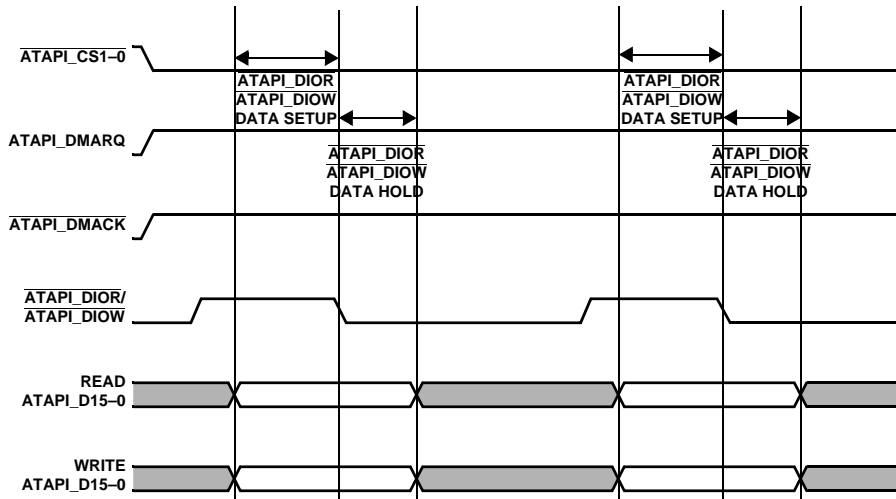


Figure 24-9. Sustaining a Multiword DMA Data Burst

To terminate the data burst, the device negates the `ATAPI_DMARQ` within t_L of the assertion of the current `ATAPI_DIOR` or `ATAPI_DIOW` pulse. The last data word for the burst is then transferred by the negation of the current `ATAPI_DIOR` or `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 24-10](#).

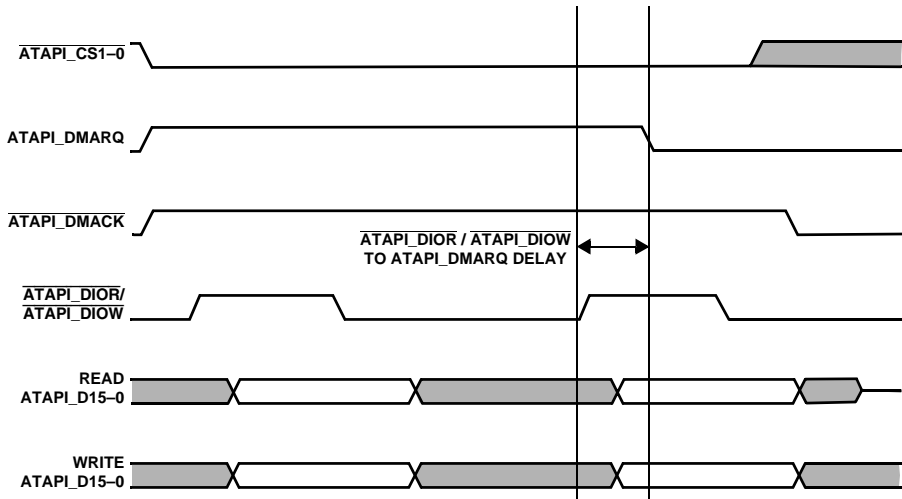


Figure 24-10. Device Terminating a Multiword DMA Burst

Functional Description

To terminate the transmission of a data burst, the host negates $\overline{\text{ATAPI_DMACK}}$ within t_j after a $\overline{\text{ATAPI_DIOR}}$ or $\overline{\text{ATAPI_DIO\overline{W}}}$ pulse. No further $\overline{\text{ATAPI_DIOR}}$ or $\overline{\text{ATAPI_DIO\overline{W}}}$ 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 $\overline{\text{ATAPI_DMACK}}$ or negates ATAPI_DMARQ at any time after detecting that $\overline{\text{ATAPI_DMACK}}$ is negated.

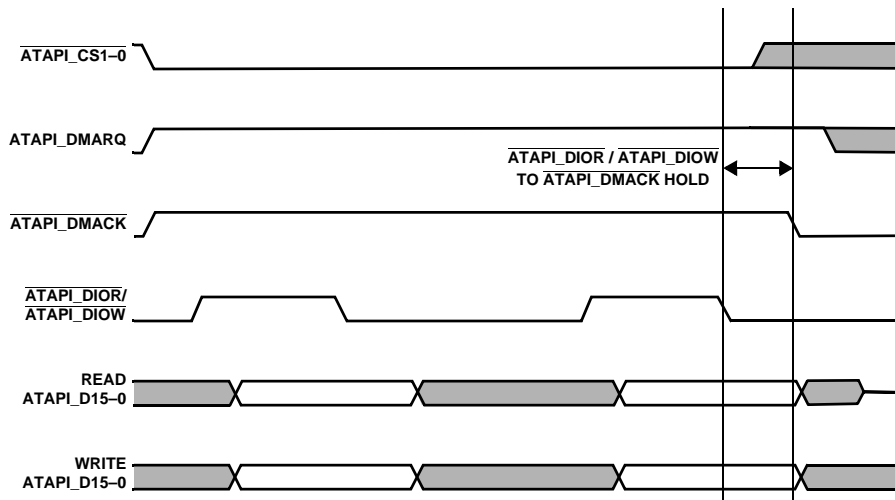


Figure 24-11. Host Terminating a Multiword DMA

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 $\overline{\text{ATAPI_DMACK}}$ in the negated state before an Ultra DMA burst is initiated.
2. The device asserts ATAPI_DMARQ to initiate an Ultra DMA burst when $\overline{\text{ATAPI_DMACK}}$ 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 $\overline{\text{ATAPI_HDMARDY}}$.
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.

Functional Description

9. The host negates $\overline{\text{ATAPI_STOP}}$ and asserts $\overline{\text{ATAPI_HDMARDY}}$ within t_{ENV} after asserting $\overline{\text{ATAPI_DMACK}}$. After negating $\overline{\text{ATAPI_STOP}}$ and asserting $\overline{\text{ATAPI_HDMARDY}}$, the host does not change the state of either signal until after receiving the first negation of $\overline{\text{ATAPI_DSTROBE}}$ 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 $\overline{\text{ATAPI_STOP}}$, 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 $\overline{\text{ATAPI_DSTROBE}}$ within t_{FS} after the host has negated $\overline{\text{ATAPI_STOP}}$ and asserted $\overline{\text{ATAPI_HDMARDY}}$. The device negates $\overline{\text{ATAPI_DSTROBE}}$ no sooner than t_{DVS} after driving the first word of data onto ATAPI_D15-0 .

In Figure 24-12, the definitions for the $\overline{\text{ATAPI_DIOW}}$, ATAPI_STOP , $\overline{\text{ATAPI_DIOR}}$, $\overline{\text{ATAPI_HDMARDY}}$, $\overline{\text{ATAPI_HSTROBE}}$, $\overline{\text{ATAPI_IORDY}}$, $\overline{\text{ATAPI_DDMARDY}}$, and $\overline{\text{ATAPI_DSTROBE}}$ signal lines are not in effect until ATAPI_DMARQ and $\overline{\text{ATAPI_DMACK}}$ are asserted.

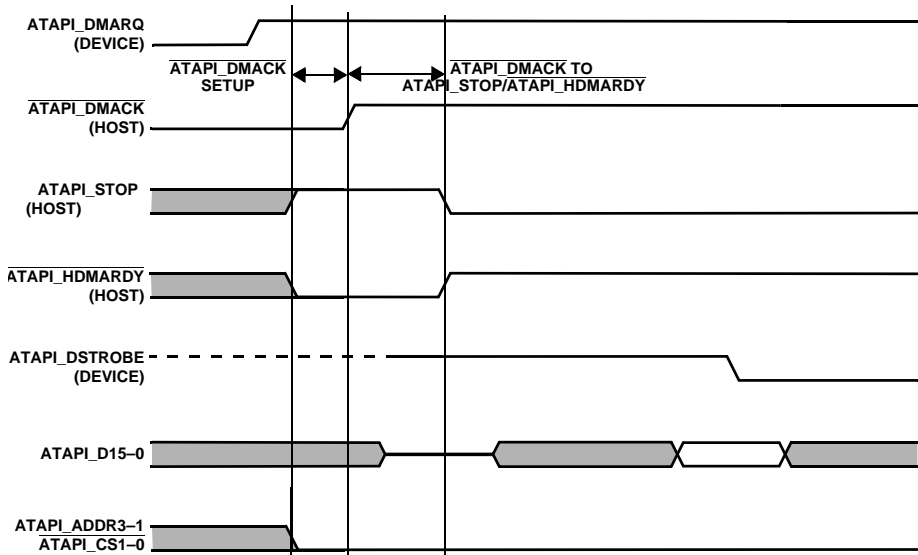


Figure 24-12. Initiating an Ultra DMA Data-In Burst

Functional Description

In [Figure 24-13](#), 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 24-32.

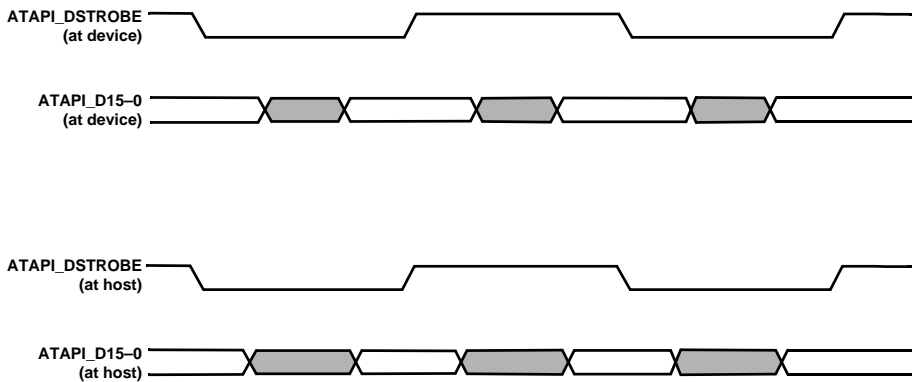


Figure 24-13. Sustaining a Ultra DMA Data IN Burst

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 $\overline{\text{ATAPI_DSTROBE}}$ edges
3. The device resumes an Ultra DMA burst by generating a $\overline{\text{ATAPI_DSTROBE}}$ 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 $\overline{\text{ATAPI_DSTROBE}}$ 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 $\overline{\text{ATAPI_HDMARDY}}$.

Functional Description

In Figure 24-14, 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.

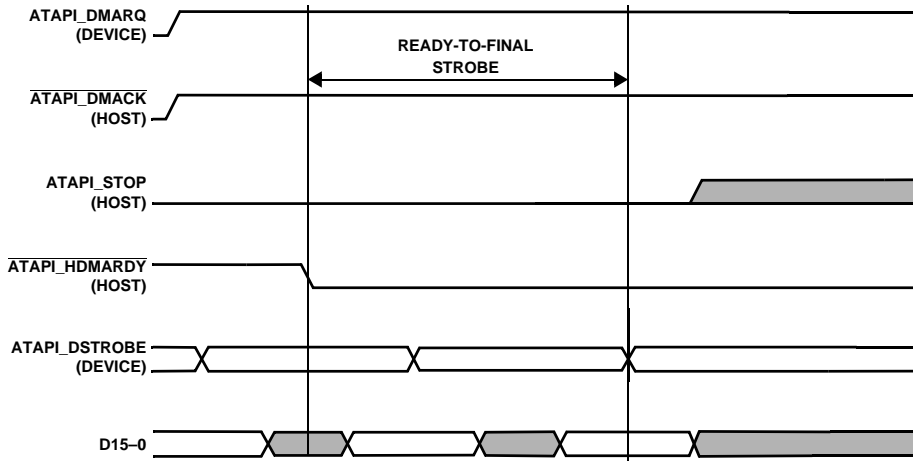


Figure 24-14. Host Pausing an Ultra DMA Data-In Burst

Ultra DMA Timing

Table 24-3. 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 $\overline{ATAPI_DMACK}$ negation) ³
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 $\overline{ATAPI_DMACK}$ to $ATAPI_STOP$ and $\overline{ATAPI_HDMARDY}$ during data-in burst initiation and from $\overline{ATAPI_DMACK}$ to $ATAPI_STOP$ during data-out burst initiation)

Functional Description

Table 24-3. Ultra DMA Sender and Recipient Timing Parameters

Name	Description
t_{RFS}	Ready-to-final-STROBE time (no STROBE edges are sent this long after negation of $\overline{ATAPI_DDMARDY}$)
t_{RP}	Ready-to-pause time (that recipient waits to pause after negating $\overline{ATAPI_DDMARDY}$)
t_{IORDYZ}	Maximum time before releasing $ATAPI_IORDY$
t_{ZIORDY}	Minimum time before driving $ATAPI_IORDY$ ⁵
t_{ACK}	Setup and hold times for $\overline{ATAPI_DMACK}$ (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
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

- 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.
- 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.
- 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.
- 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.
- 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.
- 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).
- 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 24-15, 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 “Device Terminating the Ultra DMA Data-In Transfer” on page 24-18

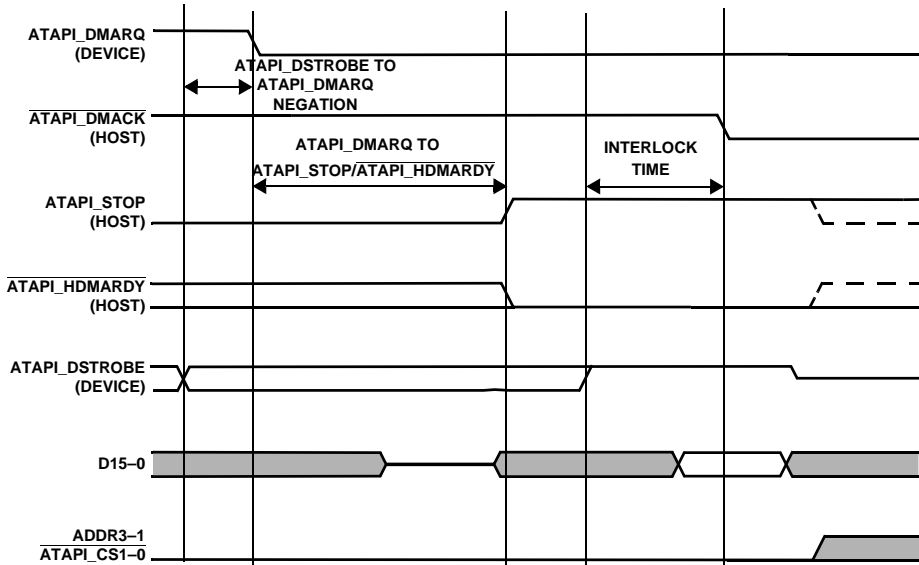


Figure 24-15. Device Terminating Ultra DMA Data-In Burst

Functional Description

In [Figure 24-16](#), 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 24-17](#).

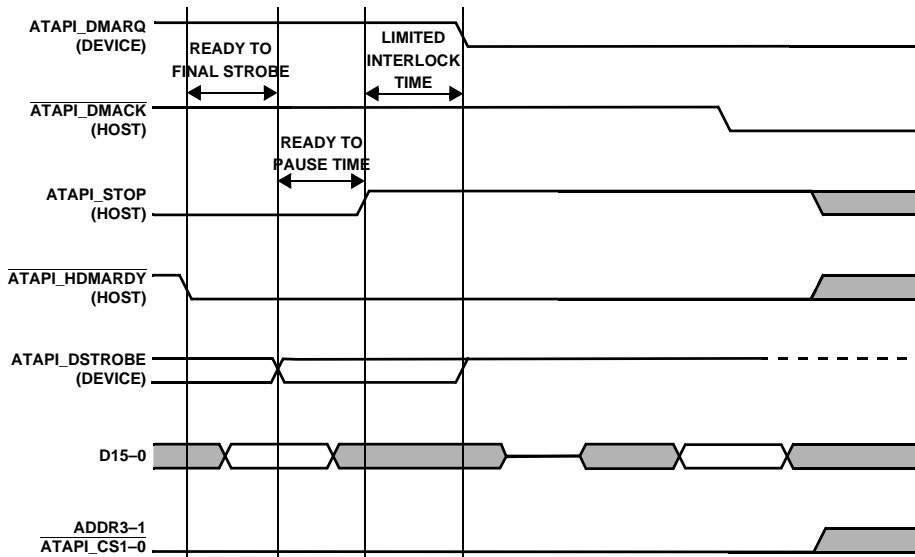


Figure 24-16. Host Terminating Ultra DMA Data-In Burst

Ultra DMA-Out Timing

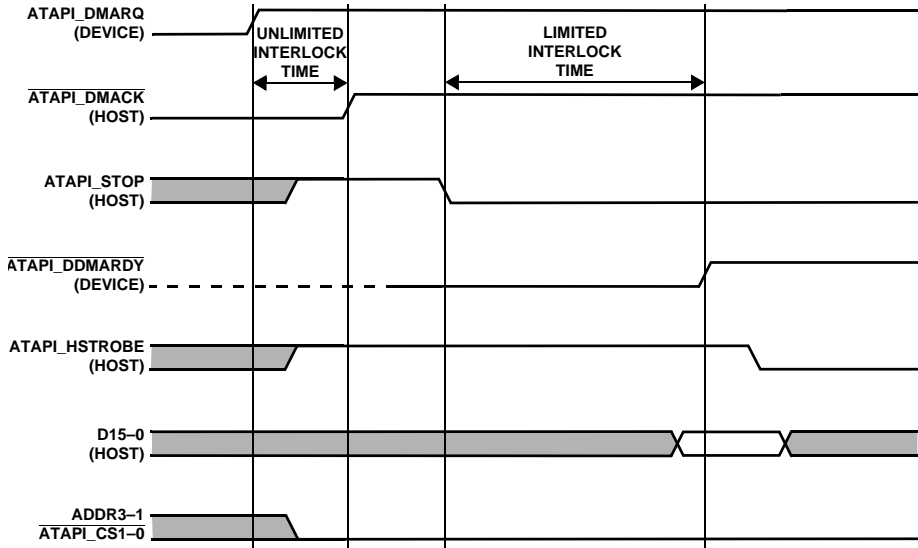


Figure 24-17. Initiating Ultra DMA Data-Out Burst

Functional Description

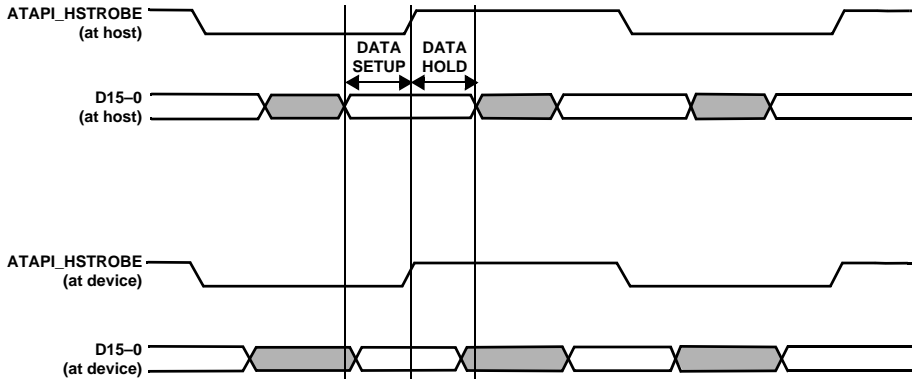


Figure 24-18. Sustaining Ultra DMA Data-Out Burst

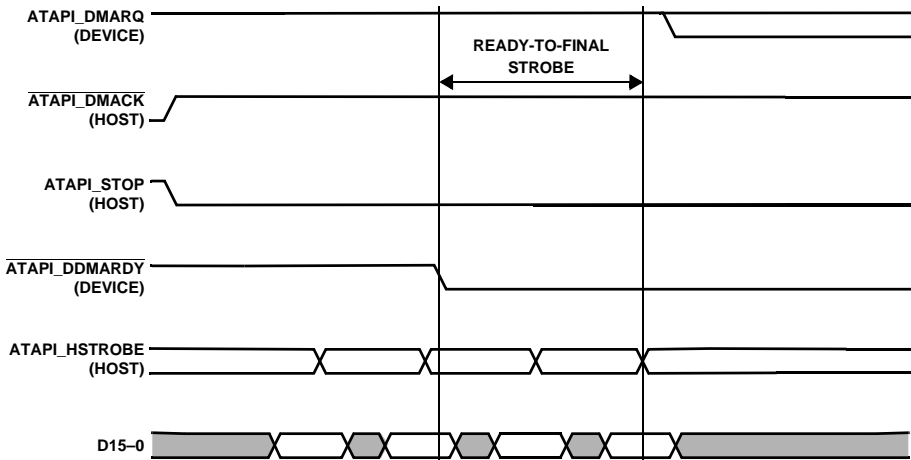


Figure 24-19. Device Pausing Ultra DMA Data-Out Burst

In Figure 24-20, 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 24-18.

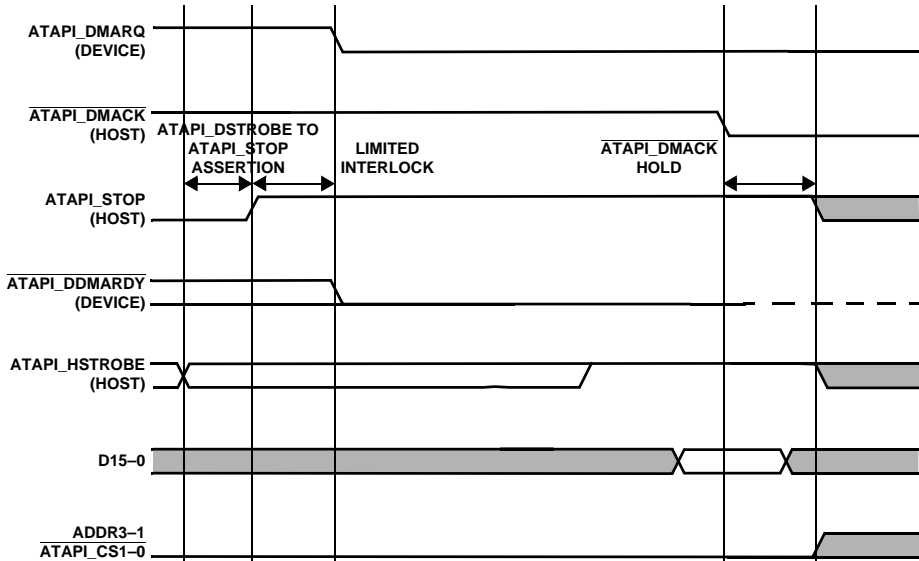


Figure 24-20. Host Terminating Ultra DMA Data-Out Burst

Functional Description

In [Figure 24-21](#), 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 24-18.

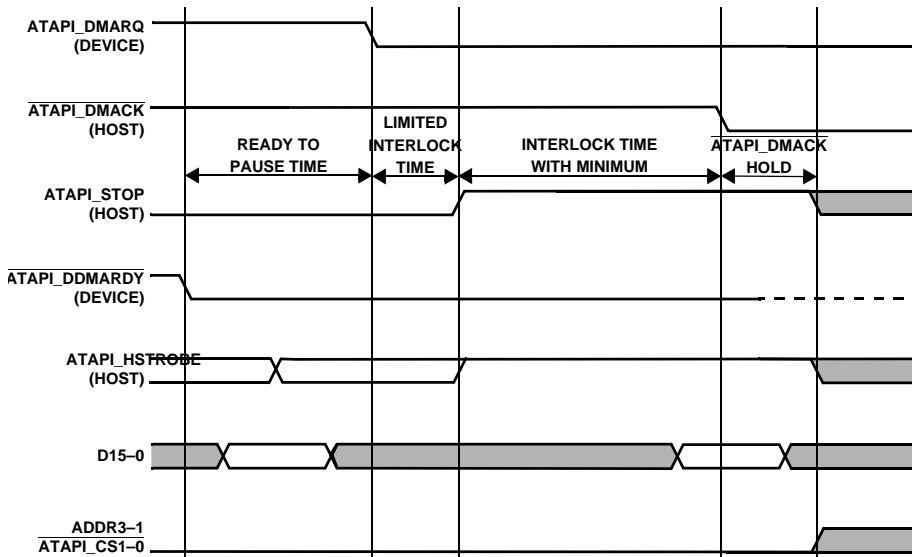


Figure 24-21. Device Terminating Ultra DMA Data OUT Burst

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.

Programming Model

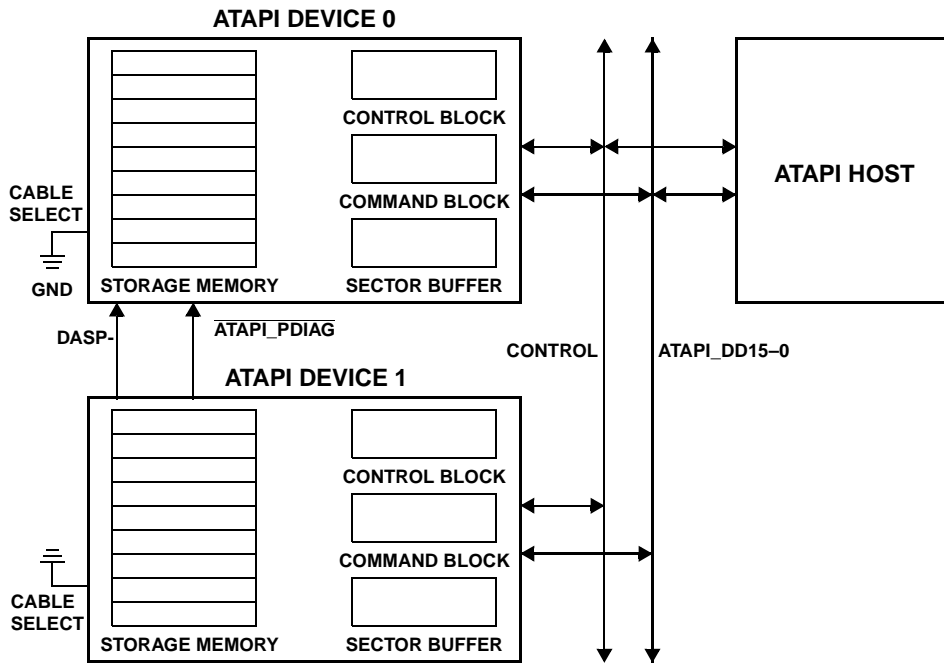


Figure 24-22. ATAPI Device and Host Configuration

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
- 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);
```

Programming Model

```
        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;
}

```

ATAPI Registers

```
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 24-4 on page 24-48](#) lists the ATAPI memory-mapped registers, starting at base address 0xFFC03800. Register addresses are given relative to the base address.

Table 24-4. ATAPI Core Registers

Register Name	ADDRESS	Description
ATAPI control registers		
ATAPI_CONTROL	0xFFC03800	ATAPI control on page 24-50
ATAPI_STATUS	0xFFC03804	ATAPI status on page 24-52
ATAPI_DEV_ADDR	0xFFC03808	ATAPI device address on page 24-53
ATAPI_DEV_TXBUF	0xFFC0380C	ATAPI device transmit buffer on page 24-54
ATAPI_DEV_RXBUF	0xFFC03810	ATAPI device receive buffer on page 24-55
ATAPI_INT_MASK	0xFFC03814	ATAPI interrupt mask on page 24-56
ATAPI_INT_STATUS	0xFFC03818	ATAPI interrupt status on page 24-57
ATAPI_XFER_LEN	0xFFC0381C	ATAPI transfer length on page 24-59
ATAPI_LINE_STATUS	0xFFC03820	ATAPI line status on page 24-60
ATAPI_SM_STATE	0xFFC03824	ATAPI state machine status on page 24-61
ATAPI_TERMINATE	0xFFC03828	ATAPI terminate on page 24-61
ATAPI_PIO_TFRCNT	0xFFC0382C	ATAPI PIO transfer count on page 24-62

Table 24-4. ATAPI Core Registers (Cont'd)

Register Name	ADDRESS	Description
ATAPI_DMA_TFRCNT	0xFFC03830	ATAPI multi-word DMA transfer count on page 24-62
ATAPI_ULTRA_IN_TFRCNT	0xFFC03834	ATAPI ultra-DMA in transfer count on page 24-63
ATAPI_ULTRA_OUT_TFRCNT	0xFFC03838	ATAPI ultra-DMA out transfer count on page 24-64
PIO and REG Mode Registers		
ATAPI_REG_TIM_0	0xFFC03840	ATAPI register transfer timing 0 on page 24-64
ATAPI_PIO_TIM_0	0xFFC03844	ATAPI programmed I/O timing 0 on page 24-65
ATAPI_PIO_TIM_1	0xFFC03848	ATAPI programmed I/O timing 1 on page 24-65
Multi-DMA mode registers		
ATAPI_MULTI_TIM_0	0xFFC03850	ATAPI multi-DMA timing 0 on page 24-66
ATAPI_MULTI_TIM_1	0xFFC03854	ATAPI multi-DMA timing 1 on page 24-66
ATAPI_MULTI_TIM_2	0xFFC03858	ATAPI multi-DMA timing 2 on page 24-67
Ultra-DMA mode registers		
ATAPI_ULTRA_TIM_0	0xFFC03860	ATAPI ultra-DMA timing 0 on page 24-67
ATAPI_ULTRA_TIM_1	0xFFC03864	ATAPI ultra-DMA timing 1 on page 24-68
ATAPI_ULTRA_TIM_2	0xFFC03868	ATAPI ultra-DMA timing 2 on page 24-68
ATAPI_ULTRA_TIM_3	0xFFC0386C	ATAPI ultra-DMA timing 3 on page 24-69

ATAPI Control and Status Registers

This section describes the details of the ATAPI core registers.

ATAPI Registers

ATAPI Control Register (ATAPI_CONTROL)

The `ATAPI_CONTROL` register (see [Figure 24-23](#)) starts, stops, and selects termination handling for ATAPI data transfers.

ATAPI Control Register (ATAPI_CONTROL)

Read/Write

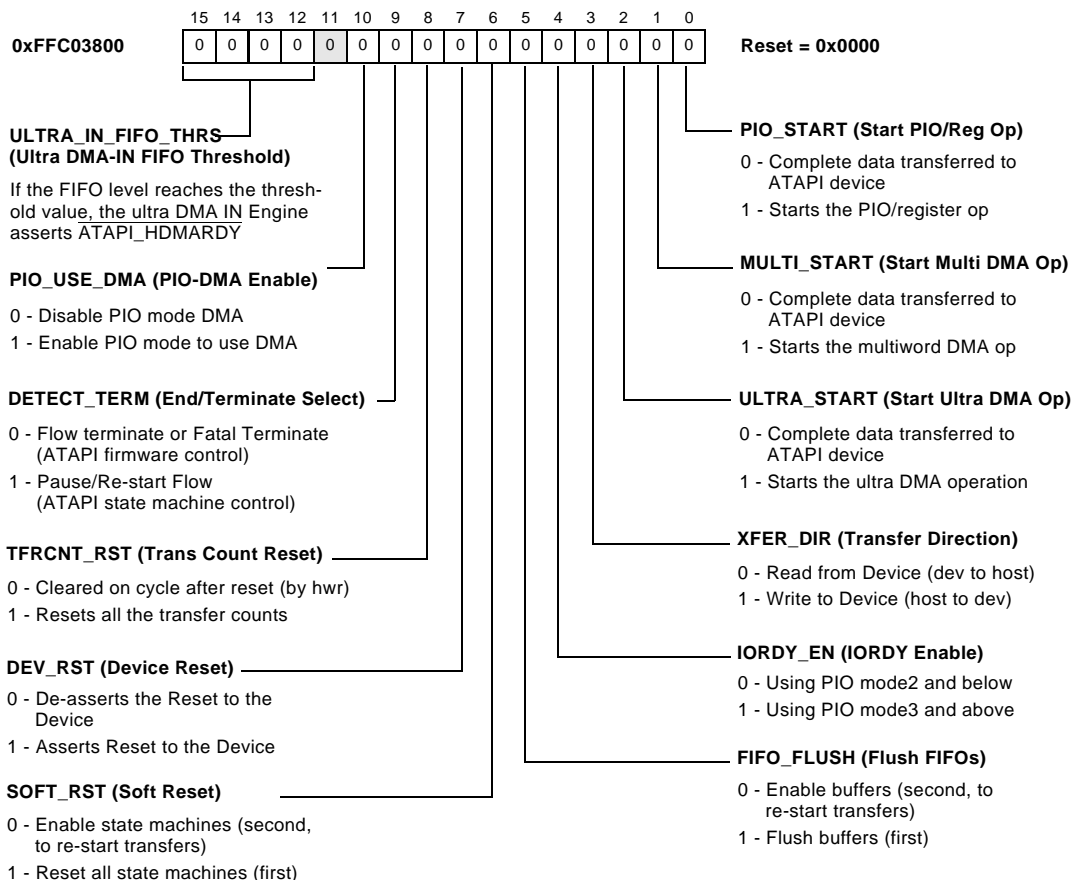


Figure 24-23. ATAPI Control Register (ATAPI_CONTROL)

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.

ATAPI Registers

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 `ATAPI_HDMARDY` pin to signal to the device to stop transferring the data.

ATAPI Status Register (`ATAPI_STATUS`)

The `ATAPI_STATUS` register (see [Figure 24-24](#)) provides status information on ATAPI data transfers in progress.

ATAPI Status Register (`ATAPI_STATUS`)

Read-only

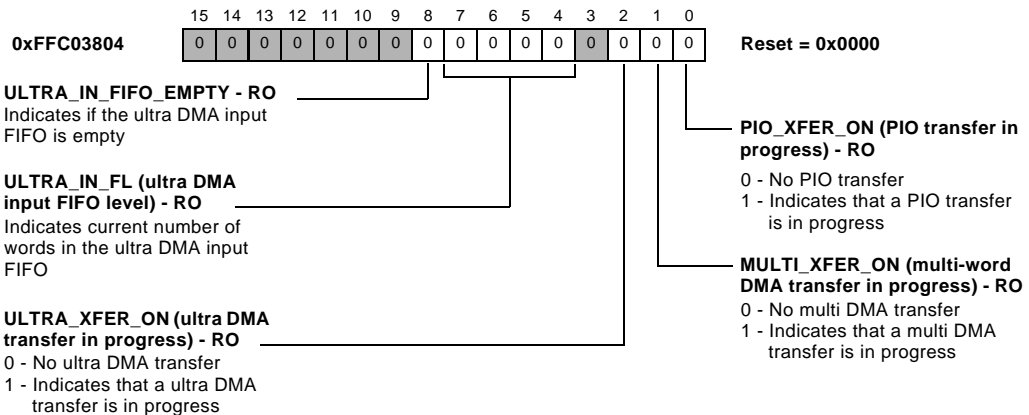


Figure 24-24. ATAPI Status Register (`ATAPI_STATUS`)

ATAPI Device Address Register (ATAPI_DEV_ADDR)

The `ATAPI_DEV_ADDR` register (see [Figure 24-25](#)) selects the ATAPI device address.

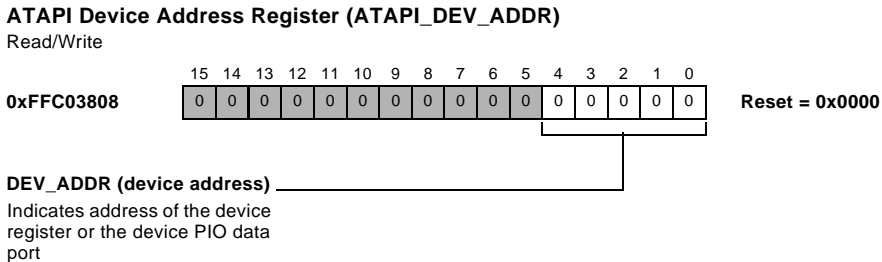


Figure 24-25. ATAPI Device Address Register (ATAPI_DEV_ADDR)

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 24-5. `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

ATAPI Registers

The ATAPI host firmware should program the `ATAPI_DEV_ADDR` register with the address of the device register, which is being accessed.

Table 24-6. `ATAPI_DEV_ADDR` Register Address Values

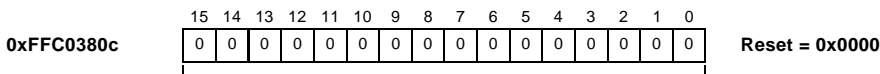
Address Value	Description
0x01	Error/Feature
0x02	Sector Count
0x03	LBA (low)
0x04	LBA (mid)
0x05	LBA (high)
0x06	Device
0x07	Status/Command
0x0E	Alternate Status/Device Control

ATAPI Device Transmit Buffer Register (`ATAPI_DEV_TXBUF`)

The `ATAPI_DEV_TXBUF` register (see [Figure 24-26](#)) holds write data for the ATAPI device register write transfers.

ATAPI Device Transmit Buffer Register (`ATAPI_DEV_TXBUF`)

Read/Write



REG_TXBUFFER (device trans buffer) _____

Write data for the device register write transfers. This register needs to be programmed with the data to be written in to the device register.

Figure 24-26. ATAPI Device Transmit Buffer (`ATAPI_DEV_TXBUF`)

ATAPI Device Receive Buffer Register (ATAPI_DEV_RXBUF)

The `ATAPI_DEV_RXBUF` register (see [Figure 24-26](#)) holds receive data for the ATAPI device register read transfers.

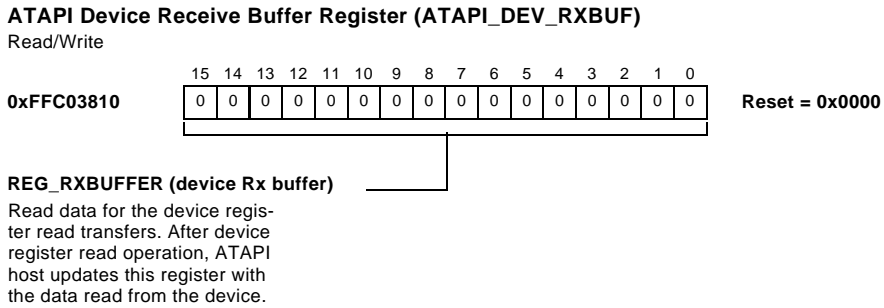


Figure 24-27. ATAPI Device Receive Buffer (ATAPI_DEV_RXBUF)

ATAPI Registers

ATAPI Interrupt Mask (ATAPI_INT_MASK) Register

The `ATAPI_INT_MASK` register (see [Figure 24-28](#)) 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 Register \(ATAPI_INT_STATUS\)](#)” on page 24-57.

ATAPI Interrupt Mask Register (ATAPI_INT_MASK)

Read/Write

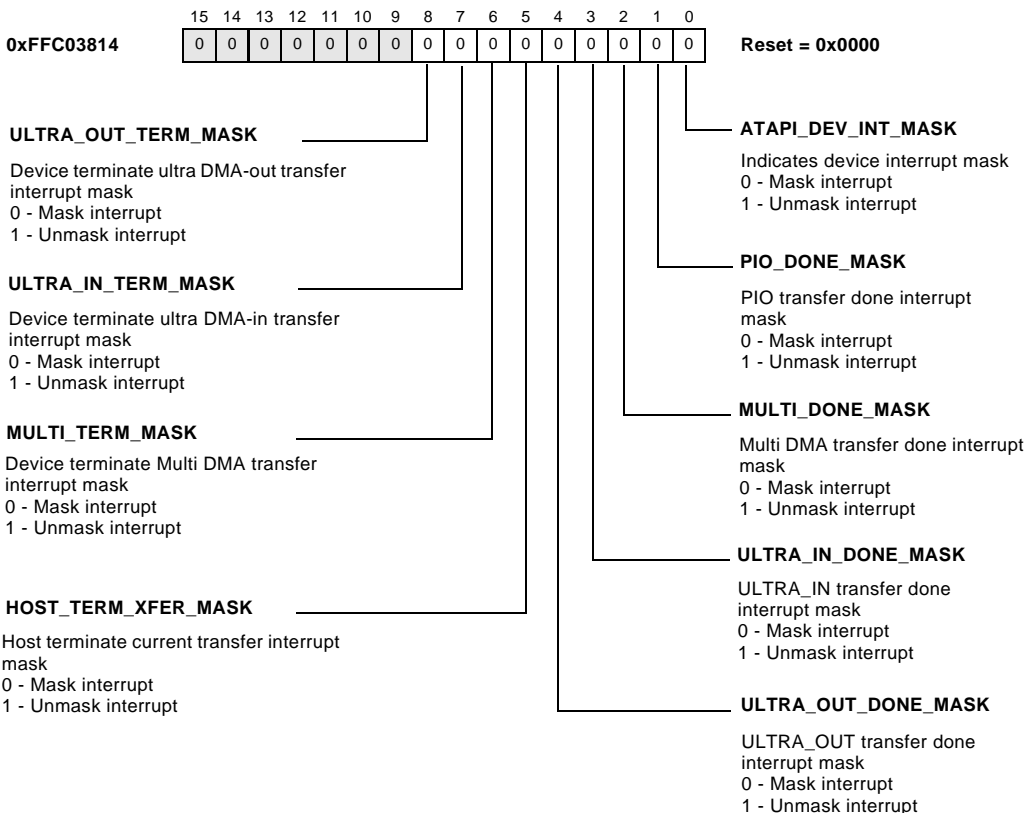


Figure 24-28. ATAPI Interrupt Mask Register (ATAPI_INT_MASK)

ATAPI Interrupt Status Register (ATAPI_INT_STATUS)

The `ATAPI_INT_STATUS` register (see [Figure 24-29](#)) contains information about functional areas that require service.

ATAPI Interrupt Status Register (ATAPI_INT_STATUS)

Read-only/Write-1-to-Clear

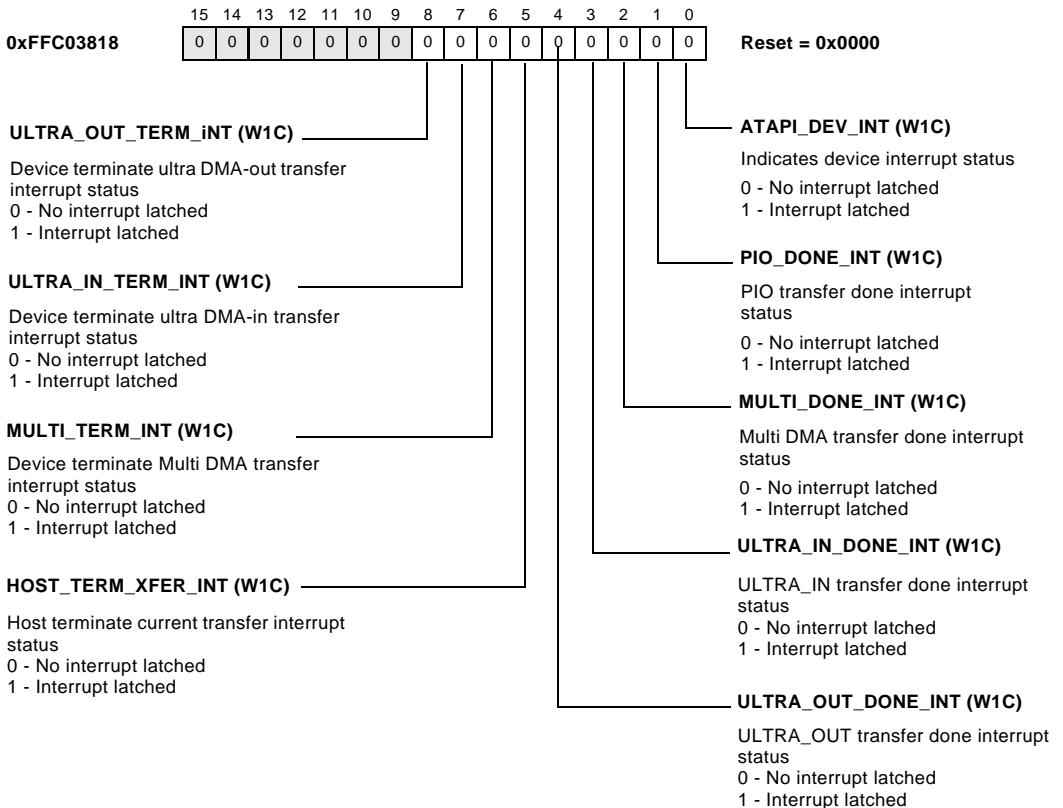


Figure 24-29. ATAPI Interrupt Status Register (ATAPI_INT_STATUS)

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

ATAPI Registers

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 24-56](#).

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.

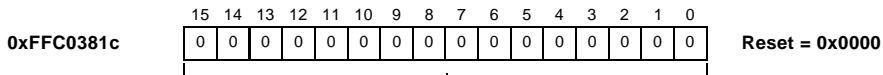
The `ULTRA_IN_TERM_INT` and `ULTRA_OUT_TERM_INT` (W1C) bits indicate that interrupts have been asserted on device termination of ultra DMA in or out transfers. The `ULTRA_IN_TERM_INT` or `ULTRA_OUT_TERM_INT` bits are set when the device initiates a termination sequence before the complete data is transferred 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 Register (ATAPI_XFER_LEN)

The `ATAPI_XFER_LEN` register (see [Figure 24-30](#)) 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.

ATAPI Transfer Length Register (ATAPI_XFER_LEN)

Read/Write



XFER_LENGTH (transfer length)

The transfer length (in number of ATA words) needs to be programmed with the number of sectors to be transferred from device to host or vice versa.

Figure 24-30. ATAPI Transfer Length Register (ATAPI_XFER_LEN)

ATAPI Registers

ATAPI Line Status Register (ATAPI_LINE_STATUS)

The `ATAPI_LINE_STATUS` register (see [Figure 24-31](#)) provides line status information on the ATAPI interface activity.

ATAPI Line Status Register (ATAPI_LINE_STATUS)

Read-only

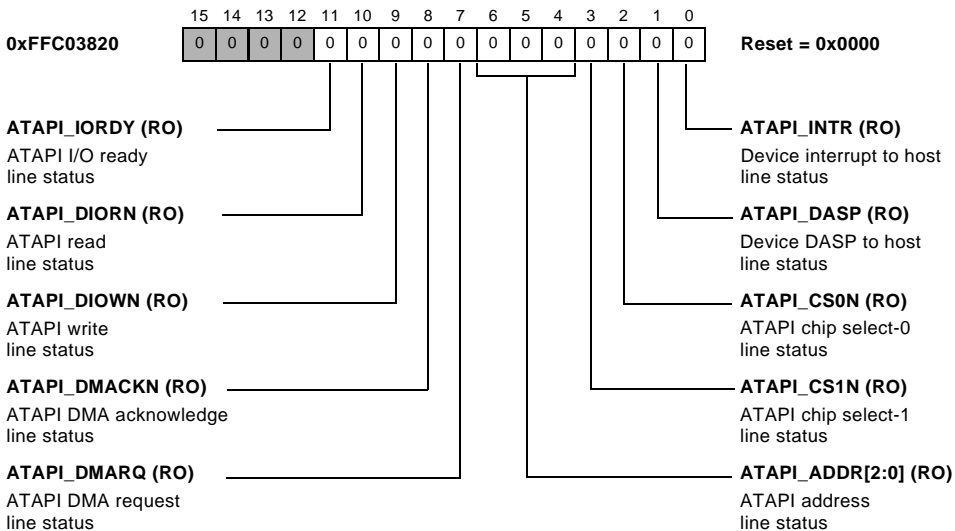


Figure 24-31. ATAPI Line Status Register (ATAPI_LINE_STATUS)

ATAPI State Machine Status Register (ATAPI_SM_STATE)

The `ATAPI_SM_STATE` register (see [Figure 24-32](#)) provides state machine status information on the ATAPI interface.

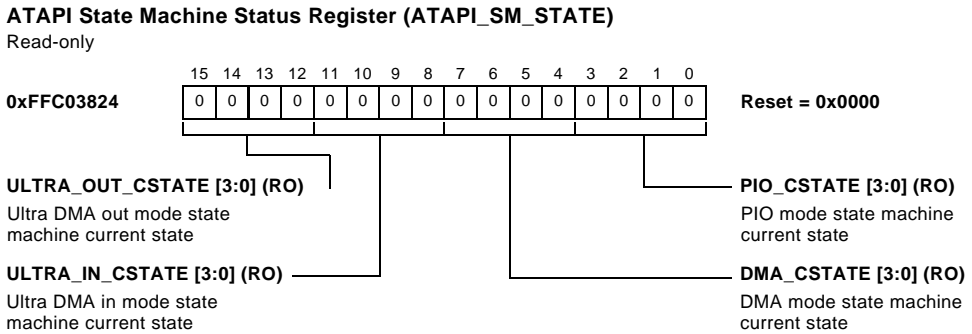


Figure 24-32. ATAPI State Machine Status Register (ATAPI_SM_STATE)

ATAPI Host Terminate Register (ATAPI_TERMINATE)

When set to 1, the `ATAPI_TERMINATE` register (see [Figure 24-33](#)) 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.

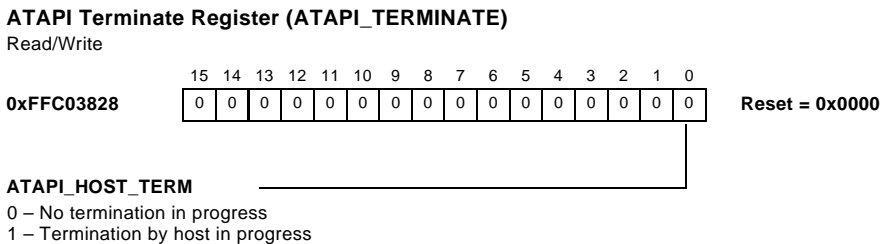


Figure 24-33. ATAPI Terminate Register (ATAPI_TERMINATE)

ATAPI Registers

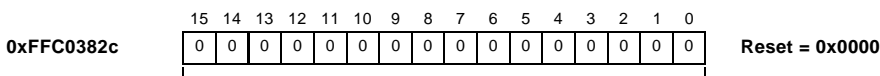
ATAPI PIO Transfer Count Register (ATAPI_PIO_TFRCNT)

The `ATAPI_PIO_TFRCNT` register (see [Figure 24-34](#)) 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.

ATAPI PIO Transfer Count Register (ATAPI_PIO_TFRCNT)

Read-only



PIO_TFRCNT (PIO trans count) - RO
PIO mode transfer count indicates the transfer count of ATA words transferred across the device for the current DMA burst in PIO mode.

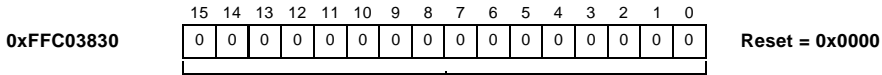
Figure 24-34. ATAPI PIO Transfer Count (ATAPI_PIO_TFRCNT)

ATAPI Multiword DMA Transfer Count (ATAPI_MULTI_TFRCNT)

The `ATAPI_MULTI_TFRCNT` register (see [Figure 24-35](#)) 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.

ATAPI DMA Transfer Count Register (ATAPI_DMA_TFRCNT)

Read-only



MULTI_TFRCNT (Multi DMA trans count) - RO

DMA mode transfer count indicates the transfer count of ATA words transferred across the device for the current DMA burst in multi DMA mode.

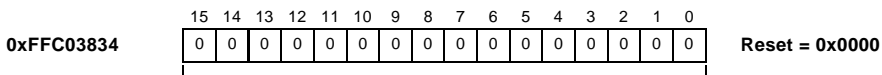
Figure 24-35. ATAPI DMA Transfer Count (ATAPI_DMA_TFRCNT)

ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT)

The `ATAPI_ULTRA_IN_TFRCNT` register (see [Figure 24-36](#)) 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.

ATAPI Ultra DMA Transfer Count Register (ATAPI_ULTRA_IN_TFRCNT)

Read-only



ULTRA_IN_TFRCNT (Ultra DMA In trans count) - RO

Ultra DMA-IN mode transfer count indicates the transfer count of ATA words transferred across the device for the current DMA burst in Ultra DMA IN mode.

Figure 24-36. ATAPI Ultra DMA Transfer Count (ATAPI_ULTRA_IN_TFRCNT) Register

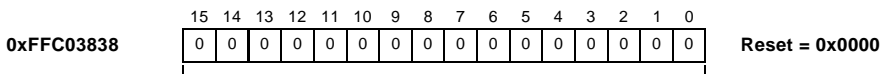
ATAPI Registers

ATAPI Ultra DMA OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT)

The `ATAPI_ULTRA_OUT_TFRCNT` register (see [Figure 24-37](#)) 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.

ATAPI Ultra DMA Transfer Count Register (ATAPI_ULTRA_OUT_TFRCNT)

Read-only



ULTRA_OUT_TFRCNT (Ultra DMA Out trans count) - RO

Ultra DMA OUT mode transfer count indicates the transfer count of ATA words transferred across the device for the current DMA burst in Ultra DMA OUT mode.

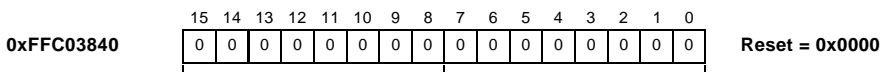
Figure 24-37. ATAPI ULTRA_OUT Transfer Count (ATAPI_ULTRA_OUT_TFRCNT)

ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0)

The `ATAPI_REG_TIM_0` register (see [Figure 24-38](#)) holds timing parameter settings (in terms of system clock counts) for register transfer operations.

ATAPI Register Transfer Timing 0 Register (ATAPI_REG_TIM_0)

Read/Write



TEOC_REG

End of cycle time for register access transfers.

T2_REG

Selects `ATAPI_DIOR` and `ATAPI_DIOW` pulsewidth.

Figure 24-38. ATAPI Register Transfer Timing 0 (ATAPI_REG_TIM_0)

ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0)

The `ATAPI_PIO_TIM_0` register (see [Figure 24-39](#)) holds timing parameter settings (in terms of system clock counts) for programmed I/O operations.

ATAPI Programmed I/O Timing 0 Register (ATAPI_PIO_TIM_0)

Read/Write

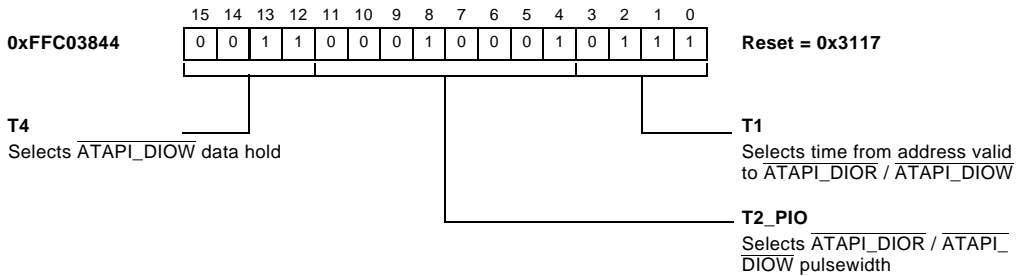


Figure 24-39. ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_0)

ATAPI Programmed I/O Timing 1 (ATAPI_PIO_TIM_1)

The `ATAPI_PIO_TIM_1` register (see [Figure 24-40](#)) holds timing parameter settings (in terms of system clock counts) for programmed I/O operations. The value of `TEOC` is `T0 - T2`.

ATAPI Programmed I/O Timing 1 Register (ATAPI_PIO_TIM_1)

Read/Write

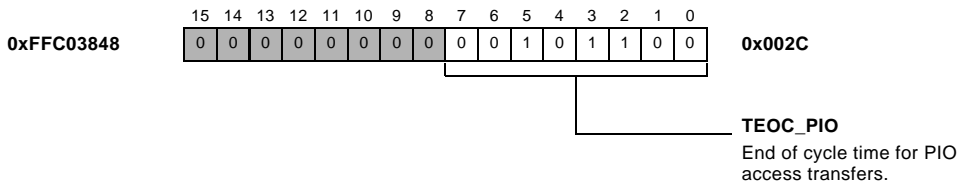


Figure 24-40. ATAPI Programmed I/O Timing 0 (ATAPI_PIO_TIM_1)

ATAPI Registers

ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0)

The `ATAPI_MULTI_TIM_0` register (see [Figure 24-41](#)) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations.

ATAPI Multi DMA Timing 0 Register (ATAPI_MULTI_TIM_0)

Read/Write

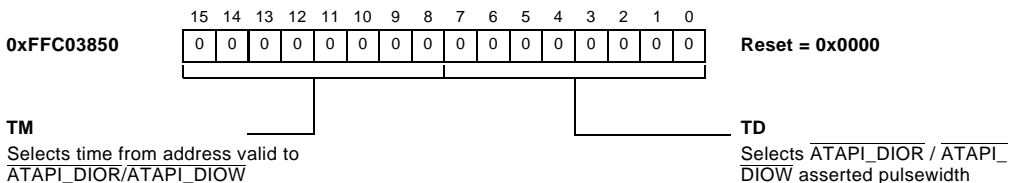


Figure 24-41. ATAPI Multi DMA Timing 0 (ATAPI_MULTI_TIM_0)

ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1)

The `ATAPI_MULTI_TIM_1` register (see [Figure 24-42](#)) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations.

ATAPI Multi DMA Timing 1 Register (ATAPI_MULTI_TIM_1)

Read/Write

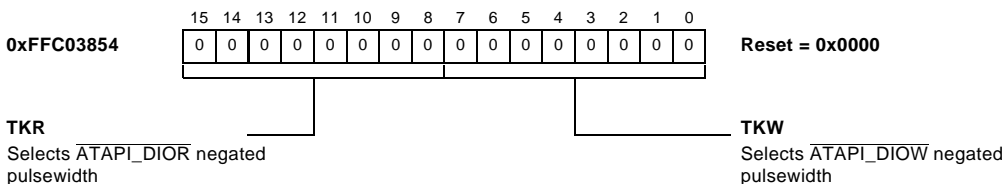


Figure 24-42. ATAPI Multi DMA Timing 1 (ATAPI_MULTI_TIM_1)

ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2)

The `ATAPI_MULTI_TIM_2` register (see [Figure 24-43](#)) holds timing parameter settings (in terms of system clock counts) for multi-word DMA operations. The value of `TEOC` is T_j .

ATAPI Multi DMA Timing 2 Register (ATAPI_MULTI_TIM_2)

Read/Write

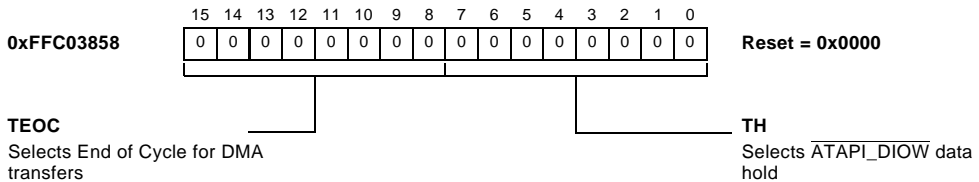


Figure 24-43. ATAPI Multi DMA Timing 2 (ATAPI_MULTI_TIM_2)

ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0)

The `ATAPI_ULTRA_TIM_0` register (see [Figure 24-44](#)) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

ATAPI Ultra DMA Timing 0 Register (ATAPI_ULTRA_TIM_0)

Read/Write

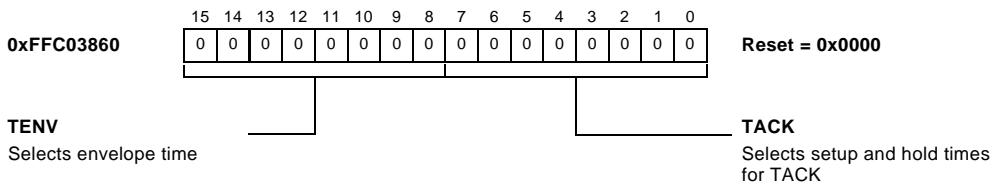


Figure 24-44. ATAPI Ultra DMA Timing 0 (ATAPI_ULTRA_TIM_0)

ATAPI Registers

ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1)

The `ATAPI_ULTRA_TIM_1` register (see [Figure 24-45](#)) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

ATAPI Ultra DMA Timing 1 Register (ATAPI_ULTRA_TIM_1)

Read/Write

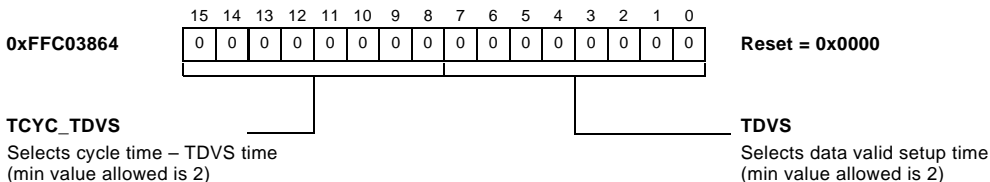


Figure 24-45. ATAPI Ultra DMA Timing 1 (ATAPI_ULTRA_TIM_1)

ATAPI Ultra DMA Timing 2 Register (ATAPI_ULTRA_TIM_2)

The `ATAPI_ULTRA_TIM_2` register (see [Figure 24-46](#)) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

ATAPI Ultra DMA Timing 2 Register (ATAPI_ULTRA_TIM_2)

Read/Write

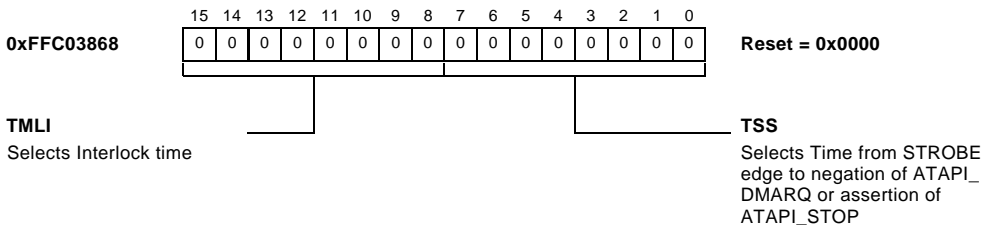


Figure 24-46. ATAPI Ultra DMA Timing 2 (ATAPI_ULTRA_TIM_2) Register

ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3) Register

The `ATAPI_ULTRA_TIM_3` register (see [Figure 24-47](#)) holds timing parameter settings (in terms of system clock counts) for ultra DMA operations.

ATAPI Ultra DMA Timing 3 Register (ATAPI_ULTRA_TIM_3)

Read/Write

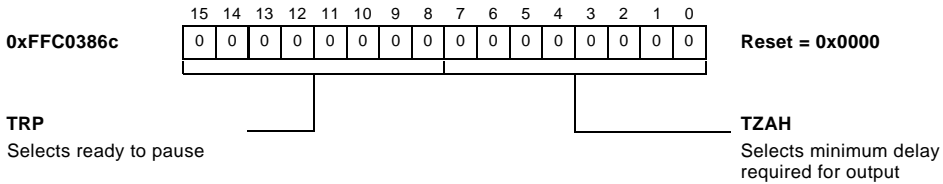


Figure 24-47. ATAPI Ultra DMA Timing 3 (ATAPI_ULTRA_TIM_3)

i 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 be used at `SCLK` frequencies lower than 100 MHz.

ATAPI Device I/O Registers

These are the registers present in an ATAPI-compliant device.

[Table 24-7](#) shows a list of ATAPI device I/O registers present on ATAPI-compliant devices.

Table 24-7. ATAPI Device I/O Registers

<code>ATAPI_CS1-0</code>	<code>ADDR3-1</code>	READ (<code>ATAPI_DIOR</code>)	WRITE (<code>ATAPI_DIOW</code>)
NN	XXX	Data Bus (Z)	Not Used
Control Block Registers			
AN	0XX	Data Bus (Z)	Not Used
AN	10X	Data Bus (Z)	Not Used

ATAPI Registers

Table 24-7. ATAPI Device I/O Registers

$\overline{\text{ATAPI_CS1-0}}$	ADDR3-1	READ ($\overline{\text{ATAPI_DIOR}}$)	WRITE ($\overline{\text{ATAPI_DIOW}}$)
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 (high 16-23)
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.

$$\overline{\text{ATAPI_CS0}} \leq \text{ADR_I (3)}$$

$$\overline{\text{ATAPI_CS1}} \leq \text{not ADR_I (3)}$$

$ADDR2-0 \leq ADR_I(2:0)$

$\overline{ATAPI_CS1-0}$ reflect the $ADR_I(3)$ signal state.

$\overline{ATAPI_CS0}$ is asserted (low level) when $ADR_I(3)$ is negated ('0').

$\overline{ATAPI_CS1}$ 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{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 24-48](#). When the Device Control register is written, both devices respond to the write regardless of which device is

ATAPI Registers

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 24-48. 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 alter-

nate 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 24-49. 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).

ATAPI Standards Reference

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. Please refer to the ATAPI specification for full details. In addition to these terms, this reference section provides:

- [“Summary of IDE/ATA Standards” on page 24-78](#)
- [“ATAPI Timing Summary” on page 24-79](#)
- [“IDE/ATA Transfer Modes and Protocols” on page 24-79](#)
- [“ATAPI Device Selection” on page 24-81](#)

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 24-8. 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

Programmed (I/O) PIO Modes

Table 24-9. 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 24-10. 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 24-11. 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 `ATAPI_PDIAG`.

ATAPI Standards Reference

When the `DEV` bit is set to 0, `DEV0` is selected. When the `DEV` bit is set to 1, `DEV1` is selected

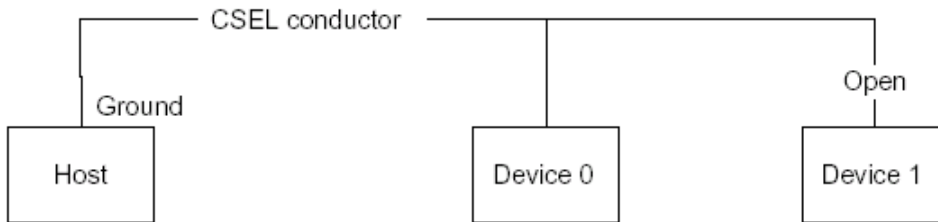


Figure 24-50. ATAPI Device Selection

25 NAND FLASH CONTROLLER

The ADSP-BF54x Blackfin processors provide a NAND flash controller (NFC) interface.

The NFC on ADSP-BF54x processors provides the hardware support for the combination of hardware and software necessary to interface ADSP-BF54x processors with NAND flash devices. The NFC provides device access timing control and hardware error checking.

This chapter includes the following sections:

- [“Overview” on page 25-2](#)
- [“Interface Overview” on page 25-4](#)
- [“Description of Operation” on page 25-5](#)
- [“Functional Description” on page 25-7](#)
- [“Programming Model” on page 25-15](#)
- [“NFC Registers” on page 25-17](#)
- [“NFC Programming Examples” on page 25-30](#)

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 25-11 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 25-1 shows the NFC interface.

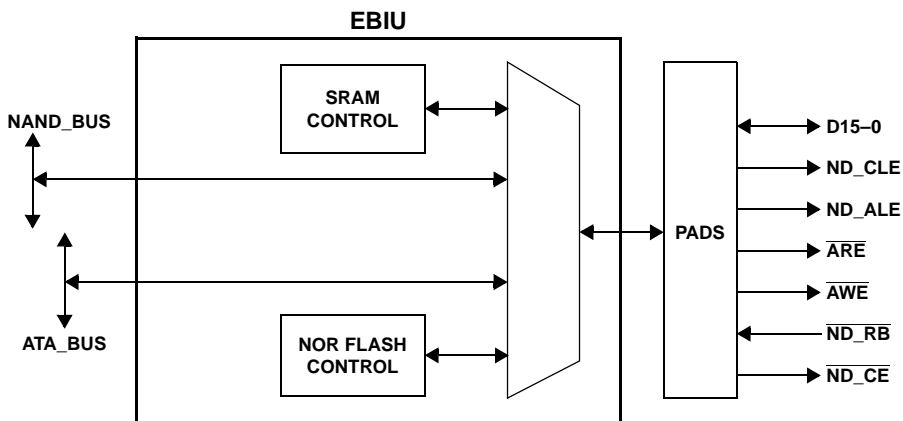


Figure 25-1. NFC Interface Block Diagram

The port pins used for NFC are shown in Table 25-1. The D15-0 bus, ND_CLE, ND_ALE, \overline{ARE} , and \overline{AWE} pins are shared with the asynchronous memory controller. In addition, the data bus D15-0 is also shared with the ATAPI peripheral.

Table 25-1. 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{ARE}	Read Enable	high	O
\overline{AWE}	Write Enable	high	O

Table 25-1. NFC External Interface (Cont'd)

Signal Name	Function	Default	Direction
$\overline{\text{ND_RB}}$	Ready/nBusy Request		I
$\overline{\text{ND_CE}}$	Chip Enable	high	O

Description of Operation

Internal Bus Interfaces

The NFC interfaces to both the PAB and DAB buses on ADSP-BF54x 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.

Description of Operation

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 25-2](#) 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 25-2. 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:

$$t_{WP} \geq \text{Max} (t_{WP_{\min}} , (t_{CS} - 1 \text{ SCLK}))$$

where t_{WP} is the time for which \overline{WE} is driven low, $t_{WP_{\min}}$ is the minimum write pulse duration from the NAND flash datasheet, and t_{CS} is the chip enable setup time from the NAND flash datasheet. See NAND Flash Controller Interface Timing in the ADSP-BF54x Blackfin Embedded Processor datasheet.

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:

$$t_{RP} > \text{Max} (t_{RP\text{min}} , t_{RE\text{Amax}} , (t_{CE\text{Amax}} - 1 \text{ SCLK}))$$

where t_{RP} is the time for which \overline{ARE} is driven low, $t_{RP\text{min}}$ is the minimum read pulse duration from the NAND flash data sheet, $t_{RE\text{Amax}}$ is the maximum read enable access time from the NAND flash datasheet, and $t_{CE\text{Amax}}$ is the maximum chip enable access time from the NAND flash datasheet. See NAND Flash Controller Interface Timing in the ADSP-BF54x Blackfin Embedded Processor datasheet.

Pin Sharing

The NFC shares the ADSP-BF54x 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 25-8](#)
- [“Page Read” on page 25-9](#)

Functional Description

- [“Additional Operations” on page 25-10](#)
- [“Write Protection” on page 25-11](#)
- [“Chip Enable Don’t Care” on page 25-11](#)
- [“NFC Error Detection” on page 25-11](#)
- [“NFC SmartMedia Support” on page 25-15](#)

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 $\overline{ND_RB}$ until the page is completely programmed. At that time, the write status bit in the NAND flash device may be checked. [Figure 25-2](#) shows the timing of a NAND flash write access for a device requiring only three address cycles.

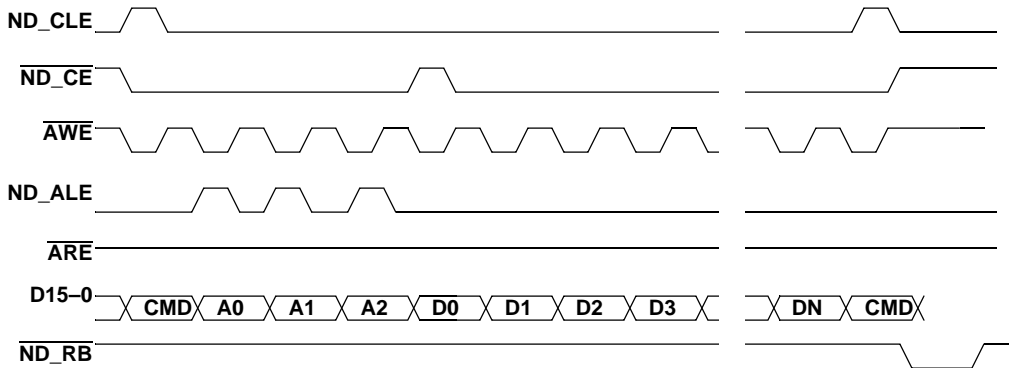


Figure 25-2. NAND Flash Program Operation

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 pro-

Functional Description

grammed. Software can compare this to the new ECC values to determine if any bit errors have occurred. [Figure 25-3](#) shows the timing of a NAND flash read access for a device requiring only three address cycles.

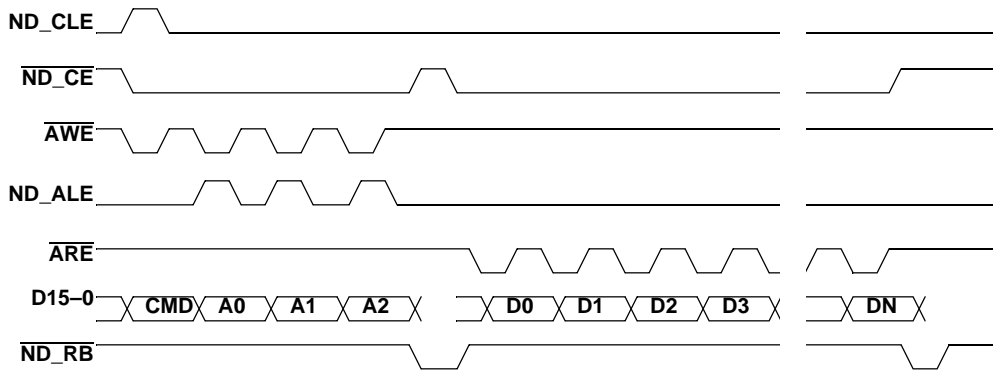


Figure 25-3. NAND Flash Read Operation

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, $\overline{ND_RB}$ may be polled in the `NFC_STAT` register or used to trigger an interrupt. Software must always poll or wait for the $\overline{ND_RB}$ 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 datasheets 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.

Functional Description

For every 256 bytes of data, 22 bits of ECC parity data are generated as follows:

$$P1 = D[1] \wedge D[3] \wedge D[5] \wedge D[7] \wedge D[9] \dots \wedge D[2047];$$

$$P2 = D[2] \wedge D[3] \wedge D[6] \wedge D[7] \wedge D[10] \wedge D[11] \dots \wedge D[2042] \wedge D[2043] \wedge D[2046] \wedge D[2047];$$

$$P4 = D[4] \wedge D[5] \wedge D[6] \wedge D[7] \wedge D[12] \wedge D[13] \wedge D[14] \wedge D[15] \wedge D[20] \wedge D[21] \wedge D[22] \wedge D[23] \dots \wedge D[2044] \wedge D[2045] \wedge D[2046] \wedge D[2047];$$

$$P8 = D[8] \wedge D[9] \wedge D[10] \wedge D[11] \wedge D[12] \wedge D[13] \wedge D[14] \wedge D[15] \wedge D[24] \wedge D[25] \wedge D[26] \wedge D[27] \wedge D[28] \wedge D[29] \wedge D[30] \wedge D[31] \dots \wedge D[2040] \wedge D[2041] \wedge D[2042] \wedge D[2043] \wedge D[2044] \wedge D[2045] \wedge D[2046] \wedge D[2047];$$

...

...

$$P1' = D[0] \wedge D[2] \wedge D[4] \wedge D[6] \wedge D[8] \dots \wedge D[2046];$$

$$P2' = D[0] \wedge D[1] \wedge D[4] \wedge D[5] \wedge D[8] \wedge D[9] \dots \wedge D[2040] \wedge D[2041] \wedge D[2044] \wedge D[2045];$$

$$P4' = D[0] \wedge D[1] \wedge D[2] \wedge D[3] \wedge D[8] \wedge D[9] \wedge D[10] \wedge D[11] \wedge D[16] \wedge D[17] \wedge D[18] \wedge D[19] \dots \wedge D[2040] \wedge D[2041] \wedge D[2042] \wedge D[2043];$$

...

...

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]
```

Functional Description

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 25-3](#).

Table 25-3. ECC Syndrome Examples

Syndrome 0	Type of Value	Meaning	Action Required
0x000000	All zero	No error in data	None
0x2CCA66	Exactly 11 bits are 1, each parity and parity' pair is 1 & 0 or 0 & 1	1-bit correctable error	Correct error
0x000040	Only 1 bit is 1	ECC data was incorrect	None
0x06B35A	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 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 $\overline{ND_RB}$, which must be cleared before beginning NFC programming sequences.

Programming Model

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 $\overline{ND_RB}$.
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 `ND_RB`.

NFC Registers

The NFC has a group of memory-mapped registers (MMRs) that regulate its operation. These registers are listed in [Table 25-4](#).

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.

NFC Registers

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 25-4](#) lists all of the NFC memory-mapped registers.

Table 25-4. NFC Memory-Mapped Registers

Register Name	Address	Description
<code>NFC_CTL</code>	0xFFC0 3B00	NFC control register on page 25-19
<code>NFC_STAT</code>	0xFFC0 3B04	NFC status register on page 25-20
<code>NFC_IRQSTAT</code>	0xFFC0 3B08	NFC interrupt status register on page 25-21
<code>NFC_IRQMASK</code>	0xFFC0 3B0C	NFC interrupt mask register on page 25-23
<code>NFC_ECC0</code>	0xFFC0 3B10	NFC ECC register 0 on page 25-23
<code>NFC_ECC1</code>	0xFFC0 3B14	NFC ECC register 1 on page 25-23
<code>NFC_ECC2</code>	0xFFC0 3B18	NFC ECC register 2 on page 25-23
<code>NFC_ECC3</code>	0xFFC0 3B1C	NCF ECC register 3 on page 25-23
<code>NFC_COUNT</code>	0xFFC0 3B20	NFC count register on page 25-25
<code>NFC_RST</code>	0xFFC0 3B24	NFC reset register on page 25-25
<code>NFC_PGCTL</code>	0xFFC0 3B28	NFC page control register on page 25-26
<code>NFC_READ</code>	0xFCC0 3B2C	NFC read data register on page 25-26

Table 25-4. NFC Memory-Mapped Registers (Cont'd)

Register Name	Address	Description
NFC_ADDR	0xFFC0 3B40	NFC address register on page 25-27
NFC_CMD	0xFFC0 3B44	NCF command register on page 25-28
NFC_DATA_WR	0xFFC0 3B48	NFC data write register on page 25-29
NFC_DATA_RD	0xFFC0 3B4C	NFC data read register on page 25-29

NFC Control Register (NFC_CTL)

The NFC_CTL register (see [Figure 25-4](#)) contains timing and mode configuration fields. The read strobe delay (RD_DLY) and write strobe delay (WR_DLY) fields extend the \overline{ARE} and \overline{AWE} strobes, respectively, by the specified number of cycles. If no extension is specified, \overline{ARE} and \overline{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.

NFC Registers

NFC Control Register (NFC_CTL)

Read/Write

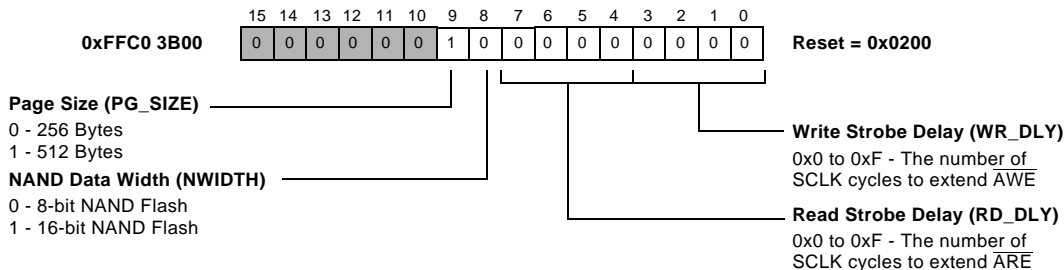


Figure 25-4. NFC Control Register (NFC_CTL)

NFC Status Register (NFC_STAT)

The `NFC_STAT` register (see [Figure 25-5](#)) contains status information. The `NBUSY` bit contains the synchronized value of the $\overline{ND_RB}$ 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.

NFC Status Register (NFC_STAT)

Read Only

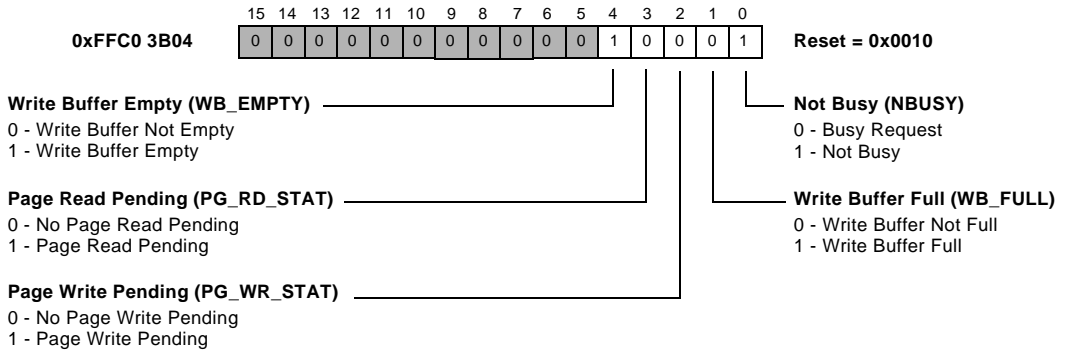


Figure 25-5. NFC Status Register (NFC_STAT)



As soon as the $\overline{\text{ND_RB}}$ signal has been enabled via PORTJ_FER and the signal is sampled as high. The NBSY bit is set in NFC_STAT and the NBSYIRQ interrupt is generated

NFC Interrupt Status Register (NFC_IRQSTAT)

The NFC_IRQSTAT register (see [Figure 25-6](#)) reports the status of additional NFC interrupt sources. All bits in this register are write-1-to-clear (W1C). The NBSYIRQ 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

NFC Registers

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.

NFC Interrupt Status Register (NFC_IRQSTAT)

Read/W1C (all bits)

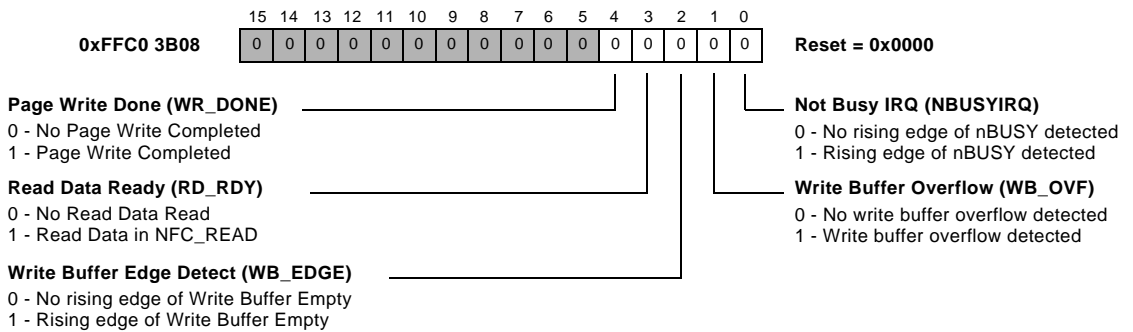


Figure 25-6. NFC Interrupt Status Register (NFC_IRQSTAT)



As soon as the $\overline{ND_RB}$ 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 25-7](#)) contains individual mask bits for each NFC interrupt source. After masking, the bits are OR'ed together and routed to the system interrupt controller.

NFC Interrupt Mask Register (NFC_IRQMASK)

Read/Write

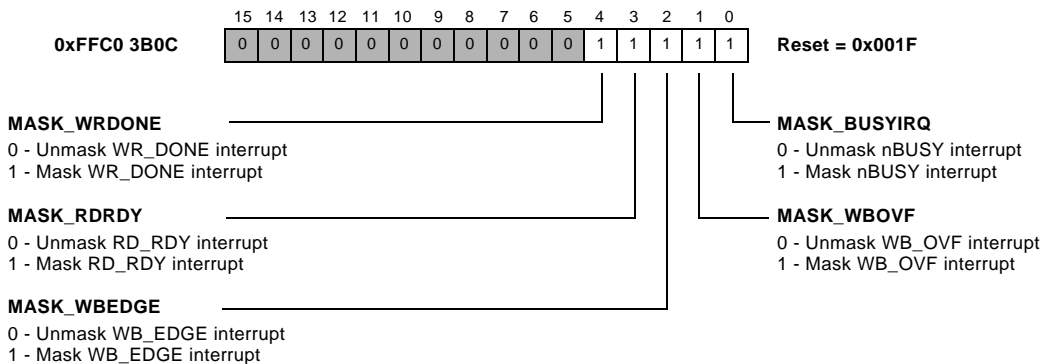


Figure 25-7. NFC Interrupt Mask Register (NFC_IRQMASK)

NFC ECC Registers (NFC_ECCx)

The `NFC_ECCx` registers (see [Figure 25-8](#)) 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.

NFC Registers

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.

NFC ECC Registers (NFC_ECCx)

Read -only

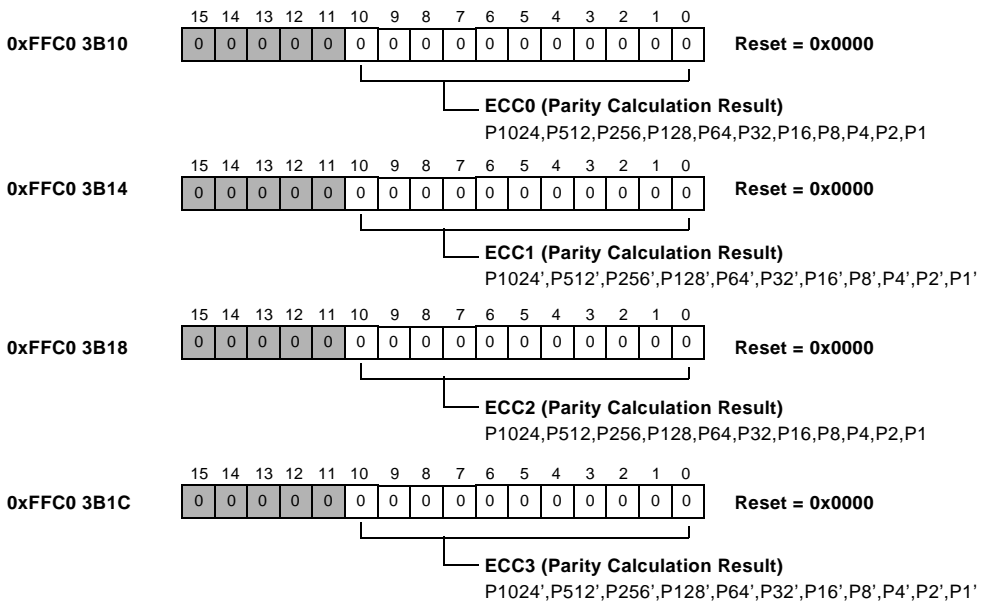


Figure 25-8. NFC ECC Registers (NFC_ECCx)

NFC Count Register (NFC_COUNT)

The `NFC_COUNT` register (see [Figure 25-9](#)) 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.

NFC Count Register (NFC_COUNT)
Read Only

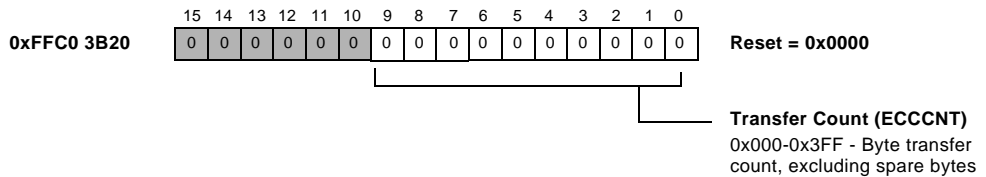


Figure 25-9. NFC Count Register (NFC_COUNT) Register

NFC Reset Register (NFC_RST)

The `NFC_RST` register (see [Figure 25-10](#)) 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.

NFC Reset Register (NFC_RST)
Read/Write

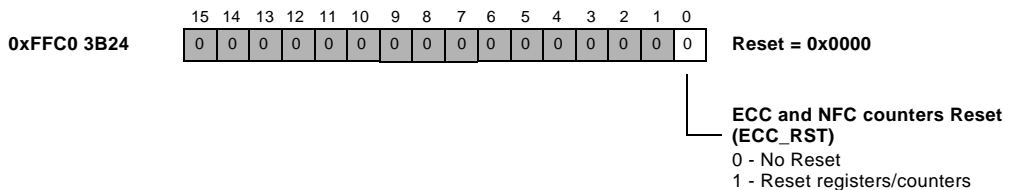


Figure 25-10. NFC Reset Register (NFC_RST)

NFC Registers

NFC Page Control Register (NFC_PGCTL)

The `NFC_PGCTL` register (see [Figure 25-11](#)) 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.

NFC Page Control Register (NFC_PGCTL)

Write-only

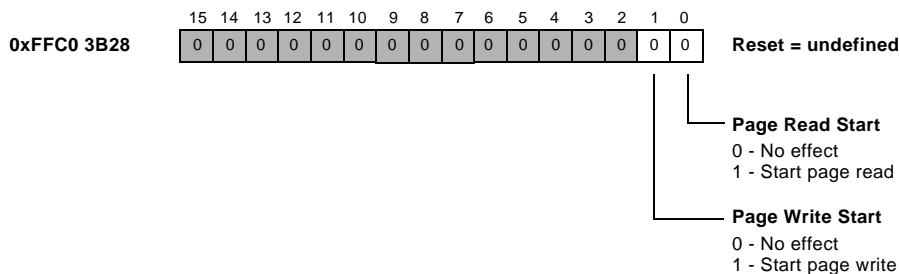


Figure 25-11. NFC Page Control Register (NFC_PGCTL)

NFC Read Data Register (NFC_READ)

The `NFC_READ` register (see [Figure 25-12](#)) 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.

NFC Data Register (`NFC_READ`)

Read-only

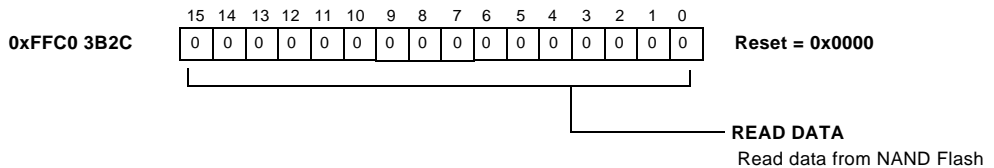


Figure 25-12. NFC Read Data Register (`NFC_READ`)

NFC Address Register (`NFC_ADDR`)

The `NFC_ADDR` register (see [Figure 25-13](#)) 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.

NFC Address Register (`NFC_ADDR`)

Write-only

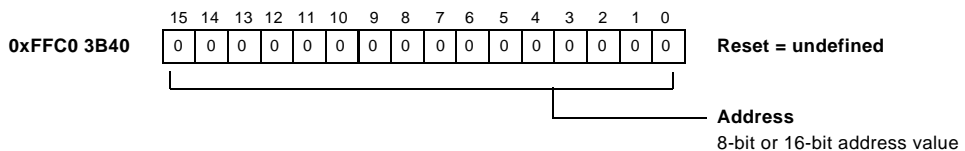


Figure 25-13. NFC Address Register (`NFC_ADDR`)

NFC Command Register (NFC_CMD)

The `NFC_CMD` register (see [Figure 25-14](#)) 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.

NFC Command Register (NFC_CMD)

Write-only

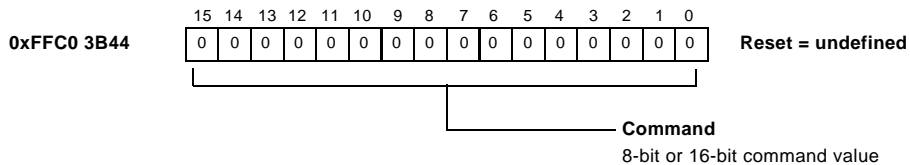


Figure 25-14. NFC Command Register (NFC_CMD)

NFC Data Write Register (NFC_DATA_WR)

The NFC_DATA_WR register (see [Figure 25-15](#)) 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.

NFC Data Write Register (NFC_DATA_WR)

Write-only

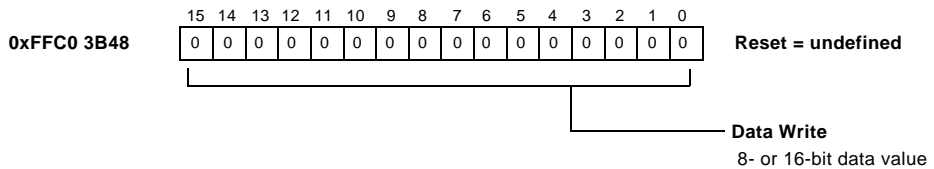


Figure 25-15. NFC Data Write Register (NFC_DATA_WR)

NFC Data Read Register (NFC_DATA_RD)

The NFC_DATA_RD register (see [Figure 25-16](#)) 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.

NFC Data Read Register (NFC_DATA_RD)

Write-only

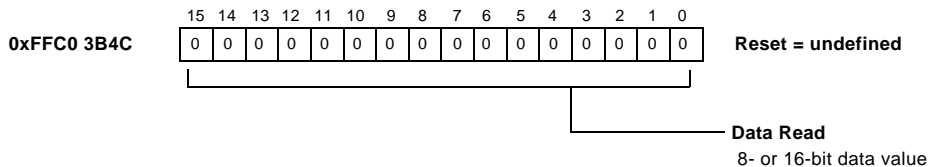


Figure 25-16. NFC Data Read Register (NFC_DATA_RD)

NFC Programming Examples

[Listing 25-1](#) illustrates an example initialization sequence to enable the use of the NFC.

Listing 25-1. 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));
R7.H = hi(MUX(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 | nWIDTH | 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;

```

NFC Programming Examples

[Listing 25-2](#) 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 25-2. 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;

/*****

```

NFC Programming Examples

```
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:

/*****

```

NFC Programming Examples

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+ 1o(NFC_READ - NFC_CTL)](z);
ssync;
w[P5+ 1o(NFC_IRQSTAT - NFC_CTL)] = R7;
_read_nfc_page_end: b[P4++] = R6;

/* Read and store the error correction parity data */
R7 = w[P5+ 1o(NFC_ECC0 - NFC_CTL)](z);
R6 = w[P5+ 1o(NFC_ECC1 - NFC_CTL)](z);
R6 <<= 11;
R7 = R7 | R6;
_read_data_end: [P3++] = R7;
```

Listing 25-3 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 con-

firmation 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 25-3. 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);

```

NFC Programming Examples

```
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:

```

NFC Programming Examples

```
_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 25-4](#) 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 25-4. 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);

```

NFC Programming Examples

```
w[P5+ 1o(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+1o(NFC_ADDR - NFC_CTL)] = R7;
w[P5+1o(NFC_ADDR - NFC_CTL)] = R7;
w[P5+1o(NFC_ADDR - NFC_CTL)] = R7;

/*****
Wait for write buffer to become empty
*****/
_check_write_buffer_empty_again:
    R6 = w[P5 + 1o(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+1o(NFC_ADDR - NFC_CTL)] = R7;
w[P5+1o(NFC_ADDR - NFC_CTL)] = R7;
R7 = 0x30(z);
w[P5+1o(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;

```

NFC Programming Examples

```
_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);

```

NFC Programming Examples

```
    ssync;  
    w[P5+ 1o(NFC_IROSTAT - NFC_CTL)] = R7;  
    _read_page_spare_end: b[P4++] = R6;
```

[Listing 25-5](#) 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 25-5. 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:

```

NFC Programming Examples

```

    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);

```

NFC Programming Examples

```
w[P5 + 1o(NFC_IRQSTAT - NFC_CTL)] = R7;

/* Read and store the error correction parity data */
R7 = w[P5+ 1o(NFC_ECC0 - NFC_CTL)](z);
R6 = w[P5+ 1o(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 = 1o(_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 + 1o(NFC_DATA_WR - NFC_CTL)] = R7;
    w[P5 + 1o(NFC_DATA_WR - NFC_CTL)] = R6;
    w[P5 + 1o(NFC_DATA_WR - NFC_CTL)] = R5;
    w[P5 + 1o(NFC_DATA_WR - NFC_CTL)] = R4;

_check_writes_spare_completed:
    R7 = w[P5 + 1o(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+ 1o(NFC_CMD - NFC_CTL)] = R7;

/*****
 Wait for the NFC not busy wakeup interrupt
*****/
_wait_for_ready:
    IDLE;
    R7= w[P5 + 1o(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+1o(NFC_IRQSTAT - NFC_CTL)] = R7;

```

NFC Programming Examples

26 ENHANCED PARALLEL PERIPHERAL INTERFACE

This chapter describes the enhanced parallel peripheral interface (EPPI) and includes the following sections:

- “Overview” on page 26-1
- “Interface Overview” on page 26-5
- “Description of Operation” on page 26-7
- “Functional Description” on page 26-11
- “EPPI Data Path Options” on page 26-29
- “Programming Model” on page 26-66
- “EPPI Registers” on page 26-76

Overview

The ADSP-BF54x 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.

Overview

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.

Enhanced Parallel Peripheral Interface

- 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 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.

Overview

To reduce pin count, some EPPI module pins are multiplexed with other EPPI pins and peripheral pins. (See [Figure 26-8 on page 26-30](#) 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 PPI_x, 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 26-1](#).

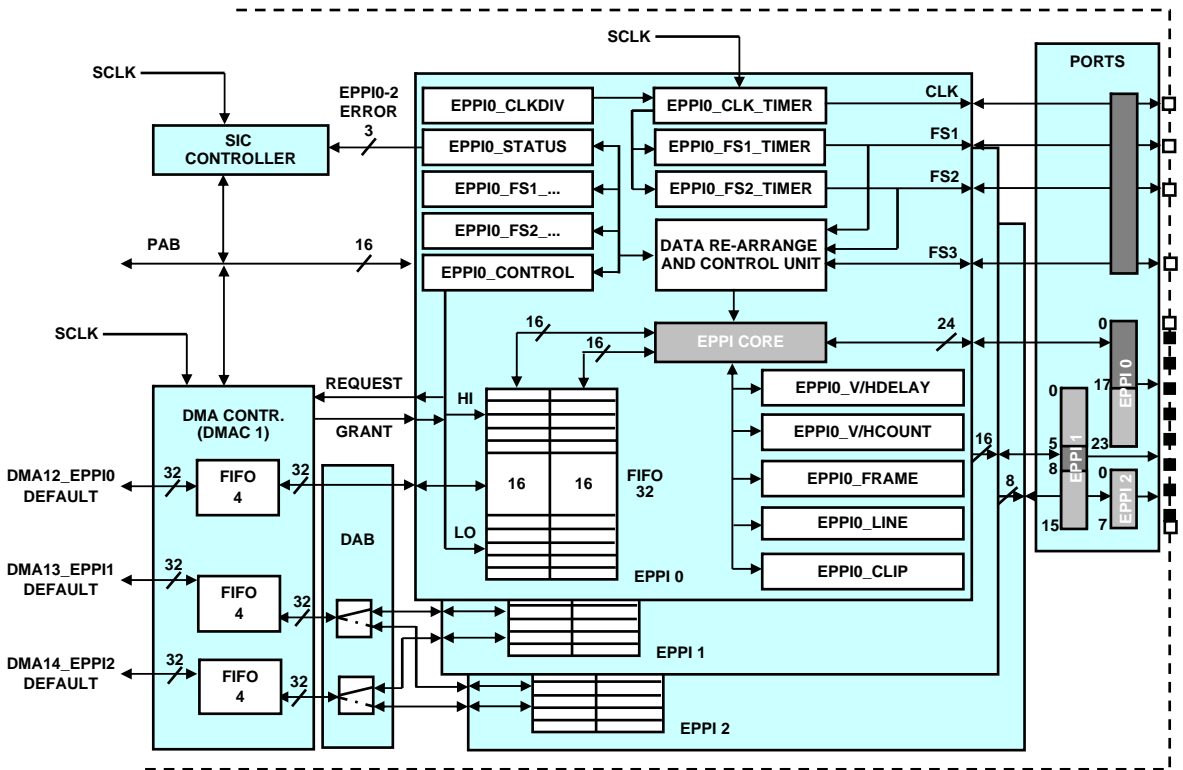


Figure 26-1. EPPI Block Diagram

Interface Overview

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 `PPIx_CLK` is $SCLK/2$. When using an external clock, the maximum frequency for `PPIx_CLK` is 75 MHz.



When using an external `PPIx_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 `PPIx_CONTROL` register is set. The generated clock frequency is then determined by the value in the `PPIx_CLKDIV` register.

Description of Operation

The following sections provide descriptions of EPPI operations.

Table 26-1. 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#11FS_ 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#11FS_ CFG=b#00		
GP 1 FS	TX	DIR=1, XFR_TYPE=11FS_C FG=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#11FS_ CFG=b#01		
GP 2 FS	TX	DIR=1, XFR_TYPE=b#11FS_ 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#11FS_ CFG=b#10		

Description of Operation

Table 26-1. Operating Modes and Generic EPPI Operation (Cont'd)

		How to configure	Useful for	How to configure in ITU R 656 TX Mode
GP 3 FS	TX	DIR=1, XFR_TYPE=b#11FS_ 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#11FS_ 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 PPIx_CLK and frame syncs are aborted.

In software, the EPPI can be reset and re-configured by writing 0 to the PPIx_EN bit in the PPIx_CONTROL register. On disabling the EPPI in this manner, only PPIx_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, PPIx_FS3 becomes a clock-gating input. This is valid for both internally and externally sourced PPIx_CLK, in both RX and TX modes. This clock gating signal must be synchronous with PPIx_CLK and must be driven by the external device on the rising edge of PPIx_CLK. Its function is to hold the sync and data lines in their current state until PPIx_FS3 is driven low. There are no additional latency cycles upon coming out of clock gating mode.



If clock gating is not required, the PPIx_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, PPIX_FS3 functions as a data enable (DEN) pin. Refer to the DEN functionality in the section [“GP 2 FS Mode” on page 26-26](#) 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.

Description of Operation

Table 26-2. 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 26-3. 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



PPIx_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 `PPIx_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 `PPIx_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

Functional Description

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 26-2](#) and [Figure 26-3](#) for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported. In this mode, the horizontal (H), vertical (V), and field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The start of active video (SAV) and end of active video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H , and EAV 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

Enhanced Parallel Peripheral Interface

Field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

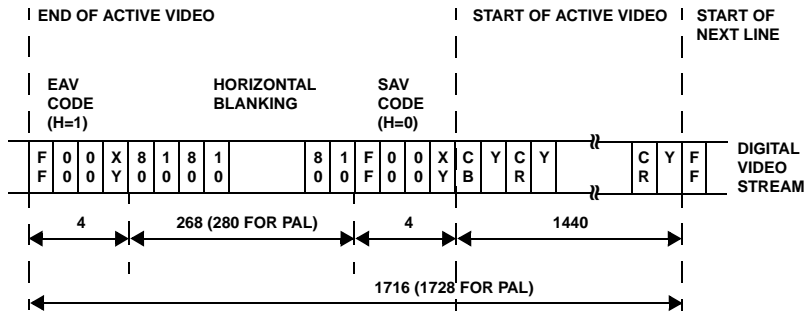


Figure 26-2. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems

Functional Description

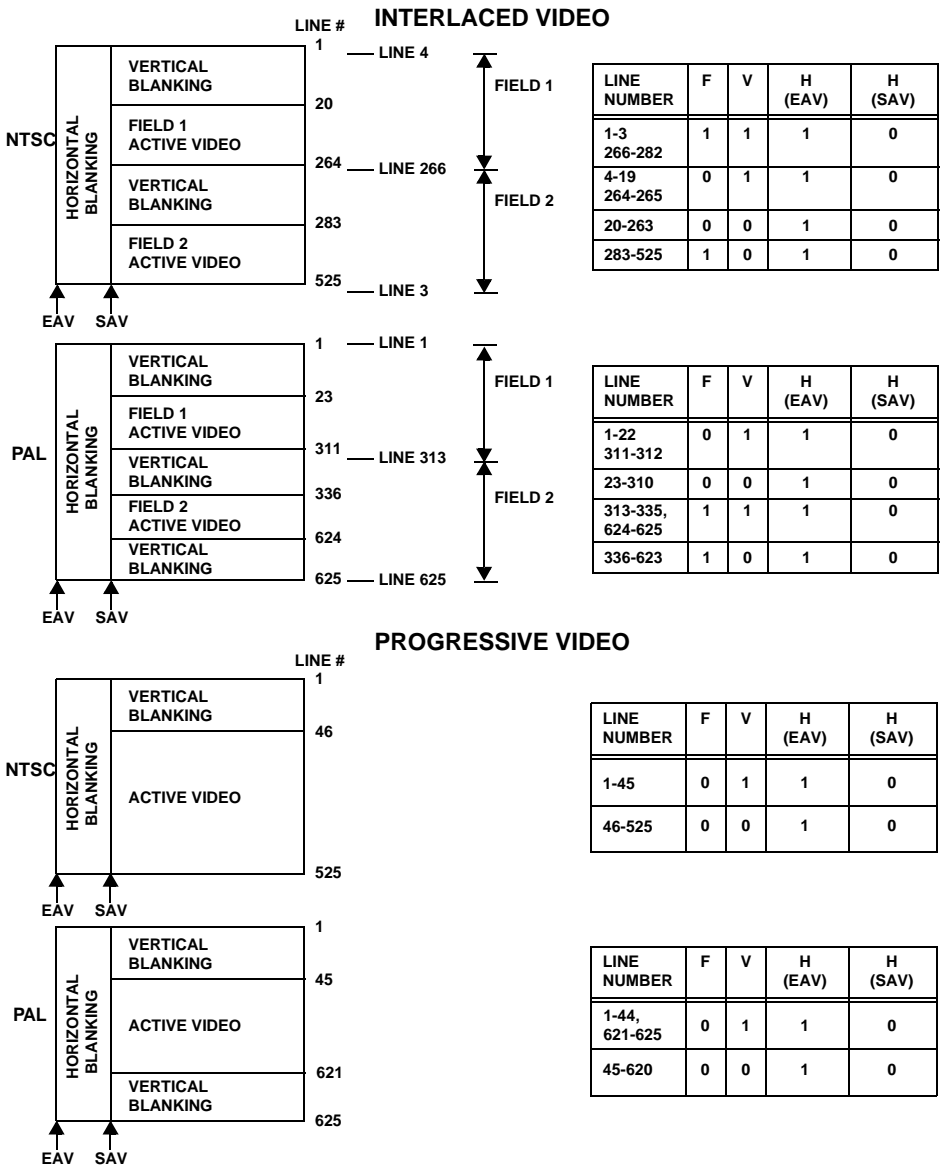


Figure 26-3. Typical Video Frame Partitioning for NTSC/PAL Systems in Interlaced and Progressive ITU-R BT.656 Systems

The SAV and EAV codes are shown in more detail in [Table 26-4](#). 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).

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

Functional Description

- $P1 = F \text{ XOR } V$
- $P0 = F \text{ XOR } V \text{ XOR } H$

Table 26-4. 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

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 `PPIX_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 `PPIX_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 26-4 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.

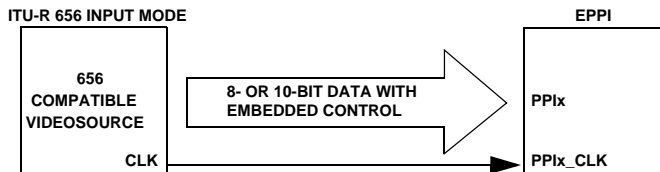


Figure 26-4. ITU-R 656 Input Modes

There are three sub-modes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 26-5 shows these three 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.

Functional Description

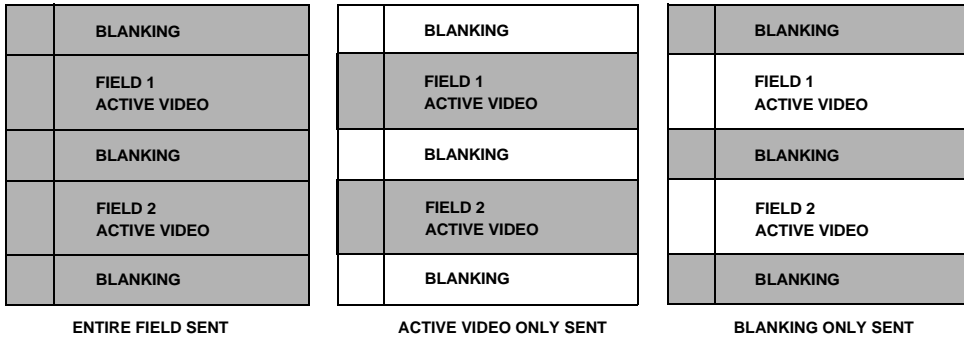


Figure 26-5. ITU-R 656 Input Sub-modes

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.


i In this mode, the user must specify the number of total (active plus vertical blanking) lines per frame in the `PPIX_FRAME` MMR, and the number of total (active plus horizontal blanking plus 8) samples per line in the `PPIX_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 `PPIX_FRAME` MMR, and the number of total (active plus horizontal blanking plus 8) samples per line in the `PPIX_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 `PPIX_CONTROL` register. The EPPI then only needs to fetch active data from memory through the DMA channel, thus saving DMA bandwidth. The `PPIX_AVPL`, `PPIX_LVB`, `PPIX_LAVF` and `PPIX_HBL` registers (shown in [Figure 26-7](#)) 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 26-6](#) 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.

Functional Description

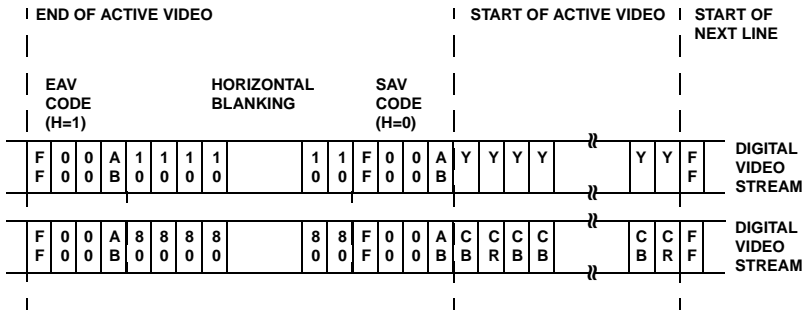


Figure 26-6. 16-Bit Transmit with Internal Blanking Generation

Enhanced Parallel Peripheral Interface

Figure 26-7 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.

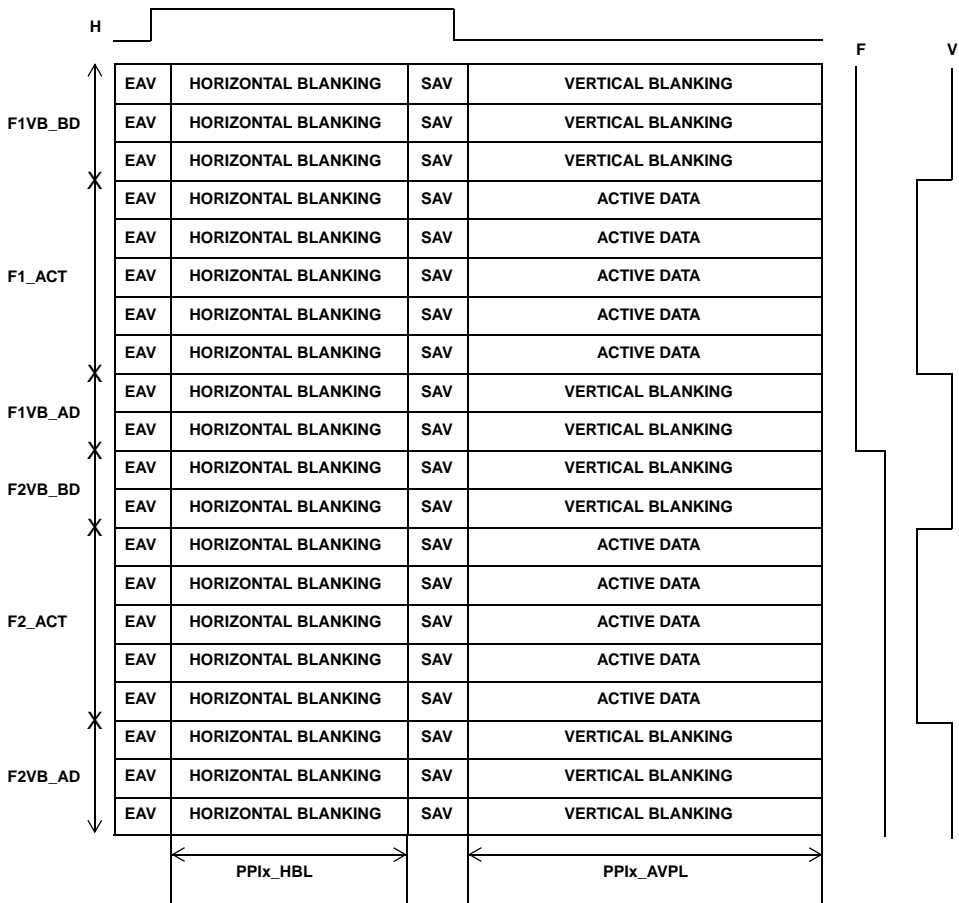


Figure 26-7. Internally Generated Blanking and Preamble Sequence with F, V, and H Signals

Functional Description

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 `PPIX_CONTROL` register. `BLANKGEN` generates preambles even in GP 2FS mode.

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 `PPIX_COUNT` registers are ignored. However, the `PPIX_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 `PPIX_FRAME`. If there is a mismatch, a frame track error is asserted in the `PPIX_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 `PPIX_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.

Functional Description

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 `PPIx_CONTROL` register.

GP 0 FS receive mode is further divided into two sub-modes: Internal Trigger (`FLD_SEL=1`) and external trigger (`FLD_SEL=0`), 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 `PPIx_LINE`, `PPIx_FRAME`, `PPIx_HCOUNT`, `PPIx_HDELAY`, `PPIx_VCOUNT` and `PPIx_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 `PPIx_FS1` and a trigger must be provided by the external device. The EPPI starts receiving the data as soon as an `PPIx_FS1` assertion is detected. After that, all subsequent data manipulation is handled by way of DMA and any activity on `PPIx_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 `PPIx_CONTROL` register. The frame sync may be provided by an external device or it can be sourced by the EPPI itself.

Functional Description

The EPPI windowing registers must be carefully programmed in GP 1 FS mode such that:

- The `PPIX_LINE` register contains the number of clock cycles expected between two assertions of `PPIX_FS1`. This is used to keep track of Line Track errors. It must be programmed before the `PPIX_HCOUNT` register.
- The `PPIX_HDELAY` register contains the number of clock cycles to wait after the assertion of `PPIX_FS1`, for example, start of frame.
- The `PPIX_HCOUNT` register contains the number of data samples to receive or transmit for each frame.

The `PPIX_FRAME`, `PPIX_VDELAY` and `PPIX_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 `PPIX_FS1` and `VSYNC` can be connected to `PPIX_FS2`.

The EPPI can be configured in GP 2 FS mode by setting `XFR_TYPE=b#11` and `FS_CFG=b#10` in the `PPIX_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 `PPIx_FRAME` register contains the number of expected lines per frame. It should be equal to the number of `PPIx_FS1` assertions expected between start of frame syncs and is used to keep track of frame track errors. It must be programmed before the `PPIx_VCOUNT` register.
- The `PPIx_LINE` register contains the number of clock cycles expected between two assertions of `PPIx_FS1`. This is used to keep track of line track errors. It must be programmed before the `PPIx_HCOUNT` register.
- The `PPIx_HDELAY` register contains the number of clock cycles to wait after the assertion of `PPIx_FS1`, for example, start of line.
- The `PPIx_HCOUNT` register contains the number of data samples to receive or transmit for each line.
- The `PPIx_VDELAY` register contains the number of lines to wait after the start of frame is detected.
- The `PPIx_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 `PPIx_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:

Functional Description

The `PPIX_FS3` pin will assert during the “active data” regions, aligned with `PPIX_CLK` according to the clock polarity (`POLC`) settings. The frame sync polarity (`POLS`) setting does not apply here -- `PPIX_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 `PPIX_FS3` pin will assert at the start of the valid data region on each line, aligned with `PPIX_CLK` according to the clock polarity (`POLC`) settings. The frame sync polarity (`POLS`) setting does not apply here -- `PPIX_FS3` will always be active high in this mode. Once asserted, `PPIX_FS3` will stay asserted for `PPIX_HCOUNT` number of clock cycles per line, and then it will de-assert. This behavior on each line will continue for the total number of lines programmed in `PPIX_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 `PPIX_HDELAY` and/or `PPIX_VDELAY` registers, the assertion of `PPIX_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 `PPIX_FS1`, `VSYNC` can be connected to `PPIX_FS2`, and `FIELD` can be connected to `PPIX_FS3`.

The EPPI can be configured in GP 3 FS mode by setting `XFR_TYPE=b#11` and `FS_CFG=b#11` in the `PPIX_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 `PPIX_FS3`. All the windowing register (`PPIX_FRAME`, `PPIX_LINE`, `PPIX_HDELAY`, `PPIX_HCOUNT`, `PPIX_VDELAY` and `PPIX_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 `PPIX_CONTROL` register.

EPPI0 can be configured for data lengths of 8, 10, 12, 14, 16, 18 or 24 bits.

EPPI1 can be configured for data lengths of 8, 10, 12, 14, or 16 bits.

EPPI2 supports only 8-bit data.

EPPI Data Path Options

The EPPI1 data pins are multiplexed with EPPI2 data pins and some EPPI0 data pins (This is shown visually in [Figure 26-8](#)). See the `PORT_MUX` register description in the “General-Purpose Ports” chapter in *ADSP-BF54x Blackfin Hardware Reference (Volume 1 of 2)*.

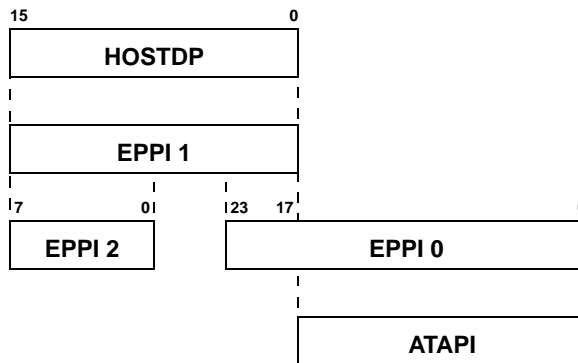


Figure 26-8. EPPI Pin Muxing

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 `PPIx_CONTROL` register.

Data Packing For Receive Modes

For receive modes, if `PACKEN` is set in `PPIx_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 `PPIX_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 26-35](#).

Data Unpacking For Transmit Modes

For transmit modes, if `PACKEN` is set in `PPIX_CONTROL`, the DMA is a 32-bit DMA and the EPPI unpacks the 32-bit word according to the `DLEN` and `SWAPEN` bit settings.

EPPI Data Path Options

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 26-35](#).

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 `PPIX_CONTROL` register is set (`SIGN_EXT=1`), then the data is sign-extended, otherwise it is zero-filled.

Split Receive Modes

The `PPIX_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 26-35.

Split Transmit Modes

The `PPIX_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.

EPPI Data Path Options

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 26-35](#).

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 PPIX_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 PPIX_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 PPIX_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 26-5](#).

Table 26-5. PPIx_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 PPIx_CONTROL register. Alternate even or odd samples may be skipped based on the SKIP_EN and SKIP_E0 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.

EPPI Data Path Options

Table 26-6 shows an 8-bit receive mode example when `PACKEN=1`.

Table 26-6. 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	0x44332211	0x11223344				
0x55						
0x66						
0x77			0x77553311		0x11335577	
0x88	0x88776655	0x55667788		0x88664422		0x22446688
0x99						
0xAA						
0xBB						
0xCC	0xCCBBAA99	0x99AABBCC				
0xDD						
0xEE						
0xFF			0xFFDDBB99		0x99BBDDFF	
0x00	0x00FFEEDD	0xDDEEFF00		0x00EECCAA		0xAACCEE00

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 26-7 shows an 8-bit receive mode example when `PACKEN=0`:

Table 26-7. 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.

EPPI Data Path Options

Table 26-8 shows a 10-Bit receive mode example when `PACKEN=1` and `SIGN_EXT=1`:

Table 26-8. 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	0xFE220111	0x0111FE22				
0x333	1			0xFF330111		0x0011FF33	
0x044	0	0x0044FF33	0xff330044		0x0044FE22		0xFE220044
0x155	0						
0x266	1	0xFE660155	0x0155FE66				
0x377	1			0xFF770155		0x0155FF77	
0x088	0	0x0088FF77	0xFF770088		0x0088FE66		0xFE660088

Enhanced Parallel Peripheral Interface

Table 26-9 Shows a 10-Bit Receive Mode Example when `PACKEN=1` and `SIGN_EXT=0`:

Table 26-9. 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	0x02220111	0x01110222				
0x333			0x03330111		0x00110333	
0x044	0x00440333	0x03330044		0x00440222		0x02220044
0x155						
0x266	0x02660155	0x01550266				
0x377			0x03770155		0x01550377	
0x088	0x00880377	0x03770088		0x00880266		0x02660088

EPPI Data Path Options

Table 26-10 shows a 10-bit receive mode example when `PACKEN=0`:

Table 26-10. 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.

Enhanced Parallel Peripheral Interface

Table 26-11 shows a 16-bit receive mode example when `PACKEN=1`.

Table 26-11. 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	0x22221111	0x11112222				
0x3333			0x33331111		0x11113333	
0x4444	0x44443333	0x33334444		0x44442222		0x22224444
0x5555						
0x6666	0x66665555	0x55556666				
0x7777			0x77775555		0x55557777	
0x8888	0x88887777	0x77778888		0x88886666		0x66668888

EPPI Data Path Options

Table 26-12 shows a 16-bit receive mode example when `PACKEN=0`:

Table 26-12. 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.

Enhanced Parallel Peripheral Interface

Table 26-13 shows an 18-bit receive mode example when `PACKEN=0`:

Table 26-13. 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
0x06666	0x00006666	0x00006666	
0x17777	0x00017777		0x00017777
0x28888	0x00028888	0x00028888	
0x39999	0x00039999		0x00039999

Table 26-14 shows an 18-bit receive mode example when `PACKEN=1`:

Table 26-14. 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
0x01122			
0x13344	0x44001122		
0x25566	0x55660133	0x66001122	
0x37788	0x03778802		0x88013344
0x099AA		0x99AA0255	
0x1BBCC	0xCC0099aa		0xBBCC0377
0x2DDEE	0xDDEE01bb	0x02DDEE00	
0x3FF12	0x03ff122d		0x03FF1201

EPPI Data Path Options

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 26-15 shows a 24-bit receive mode example when `PACKEN=0`:

Table 26-15. 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
0x66666	0x00666666	0x00666666	
0x77777	0x00777777		0x00777777
0x88888	0x00888888	0x00888888	
0x99999	0x00999999		0x00999999

Table 26-16 shows a 24-bit receive mode example when `PACKEN=1`:

Table 26-16. 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
0x112233			
0x445566	0x66112233		
0x778899	0x88994455	0x99112233	
0x00aabb	0x00AABB77		0xBB445566
0xCCDDEE		0xDDEE7788	
0xFF1234	0x34CCDDEE		0x123400AA
0x567890	0x7890FF12	0x567890CC	
0xABCDEF	0xABCDEF56		0xABCDEFFF

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.

EPPI Data Path Options

Table 26-17 shows an 8-bit split receive mode example with $SWAPEN=0$ and $SKIP_EN=0$:

Table 26-17. 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 ₁ V ₁ U ₀ V ₀	U ₁ V ₁ U ₀ V ₀			
Y ₃	Y ₃ Y ₂ Y ₁ Y ₀		Y ₃ Y ₂ Y ₁ Y ₀	Y ₃ Y ₂ Y ₁ Y ₀		Y ₃ Y ₂ Y ₁ Y ₀
V ₂						
Y ₄						
U ₂						
Y ₅						
V ₃					V ₃ V ₂ V ₁ V ₀	V ₃ V ₂ V ₁ V ₀
Y ₆						
U ₃		U ₃ V ₃ U ₂ V ₂	U ₃ V ₃ U ₂ V ₂		U ₃ U ₂ U ₁ U ₀	
Y ₇	Y ₇ Y ₆ Y ₅ Y ₄		Y ₇ Y ₆ Y ₅ Y ₄	Y ₇ Y ₆ Y ₅ Y ₄		Y ₇ Y ₆ Y ₅ Y ₄
V ₄						U ₃ U ₂ U ₁ U ₀

Enhanced Parallel Peripheral Interface

Table 26-18 shows an 8-bit split receive mode example with SWAPEN=1 and SKIP_EN=0:

Table 26-18. 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 ₀ U ₀ V ₁ U ₁	V ₀ U ₀ V ₁ U ₁			
Y ₃	Y ₀ Y ₁ Y ₂ Y ₃		Y ₀ Y ₁ Y ₂ Y ₃	Y ₀ Y ₁ Y ₂ Y ₃		Y ₀ Y ₁ Y ₂ Y ₃
V ₂						
Y ₄						
U ₂						
Y ₅						
V ₃					V ₀ V ₁ V ₂ V ₃	V ₀ V ₁ V ₂ V ₃
Y ₆						
U ₃		V ₂ U ₂ V ₃ U ₃	V ₂ U ₂ V ₃ U ₃		U ₀ U ₁ U ₂ U ₃	
Y ₇	Y ₄ Y ₅ Y ₆ Y ₇		Y ₄ Y ₅ Y ₆ Y ₇	Y ₄ Y ₅ Y ₆ Y ₇		Y ₄ Y ₅ Y ₆ Y ₇
V ₄						U ₀ U ₁ U ₂ U ₃

EPPI Data Path Options

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 26-16](#)) sent over the DMA bus is completely packed before the Luma component ($Y_4Y_5Y_6Y_7$ in [Table 26-16](#)), 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.

Enhanced Parallel Peripheral Interface

Table 26-19 shows a 16-bit split receive mode example with SWAPEN=0 and SKIP_EN=0:

Table 26-19. 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 ₀ V ₀	U ₀ V ₀			
Y ₁	Y ₁ Y ₀		Y ₁ Y ₀	Y ₁ Y ₀		Y ₁ Y ₀
V ₁					V ₁ V ₀	V ₁ V ₀
Y ₂						
U ₁		U ₁ V ₁	U ₁ V ₁		U ₁ U ₀	
Y ₃	Y ₃ Y ₂		Y ₃ Y ₂	Y ₃ Y ₂		Y ₃ Y ₂
V ₂						U ₁ U ₀

EPPI Data Path Options

Table 26-20 shows an 16-bit split receive mode example with `SWAPEN=1` and `SKIP_EN=0`:

Table 26-20. 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 ₀ U ₀	V ₀ U ₀			
Y ₁	Y ₀ Y ₁		Y ₀ Y ₁	Y ₀ Y ₁		Y ₀ Y ₁
V ₁					V ₀ V ₁	V ₀ V ₁
Y ₂						
U ₁		V ₁ U ₁	V ₁ U ₁		U ₀ U ₁	
Y ₃	Y ₂ Y ₃		Y ₂ Y ₃	Y ₂ Y ₃		Y ₂ Y ₃
V ₂						U ₀ U 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.

Enhanced Parallel Peripheral Interface

Table 26-21 shows a 16-bit split receive mode example with $SPLT_{16}=1$, $SWAPEN=0$ and $SKIP_EN=0$:

Table 26-21. 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 $PACKEN=0$.

EPPI Data Path Options

Table 26-22 shows an 8-bit transmit mode example when `PACKEN=1`:

Table 26-22. Data Sent in 8-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN=0	Pin Data when SWAPEN=1
0x11223344	0x44	0x11
0x55667788	0x33	0x22
	0x22	0x33
	0x11	0x44
	0x88	0x55
	0x77	0x66
	0x66	0x77
	0x55	0x88

Table 26-23 shows a 8-bit transmit mode example when `PACKEN=0`:

Table 26-23. 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 26-24 shows a 10-bit transmit mode example when `PACKEN=1`:

Table 26-24. Data Sent in 10-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN=0	Pin Data when SWAPEN=1
0x11112222	0x222	0x111
0x33334444	0x111	0x222
	0x044	0x333
	0x333	0x044

Table 26-25 shows a 10-bit transmit mode example when `PACKEN=0`:

Table 26-25. 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`.

EPPI Data Path Options

Table 26-26 shows a 16-bit transmit mode example when `PACKEN=1`:

Table 26-26. Data Sent in 16-bit Transmit Mode with Packing Enabled

DMA Data (32 bits)	Pin Data when SWAPEN=0	Pin Data when SWAPEN=1
0x11112222	0x2222	0x1111
0x33334444	0x1111	0x2222
	0x4444	0x3333
	0x3333	0x4444

Table 26-27 shows a 16-bit transmit mode example when `PACKEN=0`:

Table 26-27. 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 26-28](#) 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 26-28. Data Sent, 18-bit Transmit Mode with Packing Enabled

DMA Data	Pin Data (18-bits)	
	RGB_FMT_EN=0	RGB_FMT_EN=1
0x01234567	0x34567	0x08459
0x89ABCDEF	0x1EF01	0x33EC0
0x01234567	0x389AB	0x198AA
	0x12345	0x00211

[Table 26-29](#) 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 26-29. Data Sent in 18-bit Transmit Mode with Packing Disabled

DMA Data	Pin Data (18-bits)	
	RGB_FMT_EN=0	RGB_FMT_EN=1
0x01234566	0x34567	0x08459
0x89ABCDEF	0x3CDEF	0x2ACFB
0x01234567	0x34567	0x08459

EPPI Data Path Options

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 26-30 shows a 24-bit transmit mode example when `PACKEN=1`:

Table 26-30. Data Sent in 24-bit Transmit Mode

DMA Data (32 bits)	Pin Data when <code>SWAPEN=0</code>	Pin Data when <code>SWAPEN=1</code>
$R_1B_0G_0R_0$	$B_0G_0R_0$	$R_0G_0B_0$
$G_2R_2B_1G_1$	$B_1G_1R_1$	$R_1G_1B_1$
$B_3G_3R_3B_2$	$B_2G_2R_2$	$R_2G_2B_2$
	$B_3G_3R_3$	$R_3G_3B_3$

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.

Enhanced Parallel Peripheral Interface

Table 26-31 shows an 8-bit split transmit mode example with SUBSPLT_ODD=0 and SWAPEN=0:

Table 26-31. Data sent in 8-bit Split Transmit Mode with SUBSPLT_ODD=0 and SWAPEN=0

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN=0				
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 ₃ Y ₂ Y ₁ Y ₀	U ₁ V ₁ U ₀ V ₀	V ₀	U ₁ V ₁ U ₀ V ₀	V ₀
Y ₇ Y ₆ Y ₅ Y ₄	U ₃ V ₃ U ₂ V ₂	Y ₀	Y ₃ Y ₂ Y ₁ Y ₀	Y ₀
		U ₀	U ₃ V ₃ U ₂ V ₂	U ₀
		Y ₁	Y ₇ Y ₆ Y ₅ Y 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

EPPI Data Path Options

Table 26-32 shows an 8-bit split transmit mode example with SUBSPLT_ODD=1 and SWAPEN=0:

Table 26-32. Data Sent in 8-bit Split Transmit Mode with SUBSPLT_ODD=1 and SWAPEN=0

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN=0				
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

Enhanced Parallel Peripheral Interface

Table 26-33 shows an 8-bit split transmit mode example with SUBSPLT_ODD=0 and SWAPEN=1:

Table 26-33. Data Sent in 8-bit Split Transmit Mode with SUBSPLT_ODD=0 and SWAPEN=1

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 1				
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 ₃ Y ₂ Y ₁ Y ₀	U ₁ V ₁ U ₀ V ₀	U ₁	U ₁ V ₁ U ₀ V ₀	U ₁
Y ₇ Y ₆ Y ₅ Y ₄	U ₃ V ₃ U ₂ V ₂	Y ₃	Y ₃ Y ₂ Y ₁ Y ₀	Y ₃
		V ₁	U ₃ V ₃ U ₂ V ₂	V ₁
		Y ₂	Y ₇ Y ₆ Y ₅ Y 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 ₂
		Y ₄		Y ₄

EPPI Data Path Options

Table 26-34 shows an 8-bit split transmit mode example with SUBSPLT_ODD=1 and SWAPEN=1:

Table 26-34. Data Sent in 8-bit Split Transmit Mode with SUBSPLT_ODD=1 and SWAPEN=1

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 1				
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 ₃ Y ₂ Y ₁ Y ₀	V ₃ V ₂ V ₁ V ₀	V ₃	V ₃ V ₂ V ₁ V ₀	V ₃
Y ₇ Y ₆ Y ₅ Y ₄	U ₃ U ₂ U ₁ U ₀	Y ₃	Y ₃ Y ₂ Y ₁ Y ₀	Y ₃
	V ₇ V ₆ V ₅ V ₄	U ₃	U ₃ V ₃ U ₂ V ₂	U ₃
	U ₇ U ₆ U ₅ U ₄	Y ₂	Y ₇ Y ₆ Y ₅ Y 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 26-35](#) shows a 16-bit split transmit mode example with `SUBSPLT_ODD=0` and `SWAPEN=0`:

Table 26-35. Data Sent in 16-bit Split Transmit Mode with `SUBSPLT_ODD=0` and `SWAPEN=0`

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN=0				
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

EPPI Data Path Options

Table 26-36 shows a 16-bit split transmit mode example with SUBSPLT_ODD=1 and SWAPEN=0:

Table 26-36. Data Sent in 16-bit Split Transmit Mode with SUBSPLT_ODD=1 and SWAPEN=0

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN=0				
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

Enhanced Parallel Peripheral Interface

Table 26-37 shows a 16-bit split transmit mode example with SUBSPLT_ODD=0 and SWAPEN=1:

Table 26-37. Data Sent in 16-bit Split Transmit Mode with SUBSPLT_ODD=0 and SWAPEN=1

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN = 1				
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 ₁ Y ₀	V ₀ U ₀	V ₀	V ₀ U ₀	V ₀
Y ₃ Y ₂	V ₁ U ₁	Y ₁	Y ₁ Y ₀	Y ₁
		U ₀	V ₁ U ₁	U ₀
		Y ₀	Y ₃ Y ₂	Y ₀
		V ₁		V ₁
		Y ₃		Y ₃
		U ₁		U ₁
		Y ₂		Y ₂

EPPI Data Path Options

Table 26-38 shows an 16-bit split transmit mode example with `SUBSPLT_ODD=1` and `SWAPEN=1`:

Table 26-38. Data Sent in 16-bit Split Transmit Mode with `SUBSPLT_ODD=1` and `SWAPEN=1`

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN = 1				
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.

Enhanced Parallel Peripheral Interface

Table 26-39 shows a 16-bit split transmit mode example with SUBSPLT_ODD=0 and SWAPEN=0:

Table 26-39. Data Sent in 16-bit Split Transmit Mode with SPLT_16=1, SUBSPLT_ODD=0 and SWAPEN=0

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 0 SWAPEN=0				
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 ₃ Y ₂ Y ₁ Y ₀	U ₁ V ₁ U ₀ V ₀	V ₀ Y ₀	U ₁ V ₁ U ₀ V ₀	V ₀ Y ₀
Y ₇ Y ₆ Y ₅ Y ₄	U ₃ V ₃ U ₂ V ₂	U ₀ Y ₁	Y ₃ Y ₂ Y ₁ Y ₀	U ₀ Y ₁
		V ₁ Y ₂	U ₃ V ₃ U ₂ V ₂	V ₁ Y ₂
		U ₁ Y ₃	Y ₇ Y ₆ Y ₅ Y ₄	U ₁ Y ₃
		V ₂ Y ₄		V ₂ Y ₄
		U ₂ Y ₅		U ₂ Y ₅
		V ₃ Y ₆		V ₃ Y ₆
		U ₃ Y ₇		U ₃ Y ₇

Programming Model

Table 26-40 shows a 16-bit split transmit mode example with `SPLT_16=1`, `SUBSPLT_ODD=1` and `SWAPEN=0`:

Table 26-40. Data Sent in 16-bit Split Transmit Mode with `SPLT_16=1`, `SUBSPLT_ODD=1` and `SWAPEN=0`

SPLT_EVEN_ODD = 1 SUBSPLT_ODD = 1 SWAPEN=0				
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 additional information about the DMA engine, including default EPPI DMA channel mappings, refer to the “Direct Memory Access” chapter in *ADSP-BF54x Blackfin Hardware Reference (Volume 1 of 2)*.

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 26-8 on page 26-30](#).

The EPPI DMA channels can be configured for either transmit or receive operation, and have a maximum throughput of $(PPIx_CLK) \times (32 \text{ bits/transfer})$. 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,

Programming Model

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 DMA_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 details regarding configuration of DMA, refer to the “Direct Memory Access” chapter in ADSP-BF54x *Blackfin Hardware Reference*.

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 $PPIx_CONTROL$.

In addition, there are two sets of DMA Watermark levels to be programmed in the $PPIx_CONTROL$ 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 26-9](#)):

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:

Programming Model

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.

Enhanced Parallel Peripheral Interface

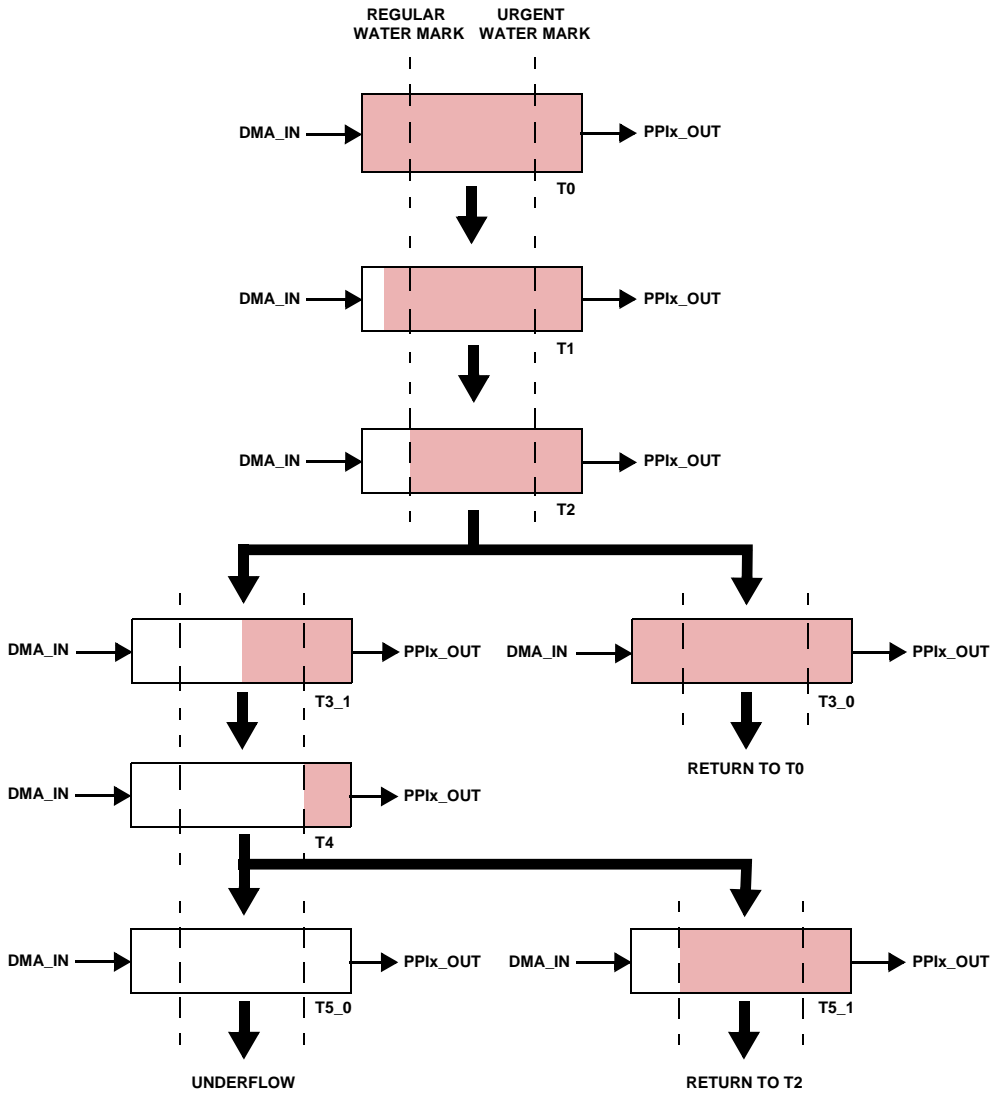


Figure 26-9. FIFO States during Transmit Modes

Programming Model

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 26-10](#)):

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.

Enhanced Parallel Peripheral Interface

For receive modes, the value of the Regular Watermark should be less than that of the Urgent Watermark.

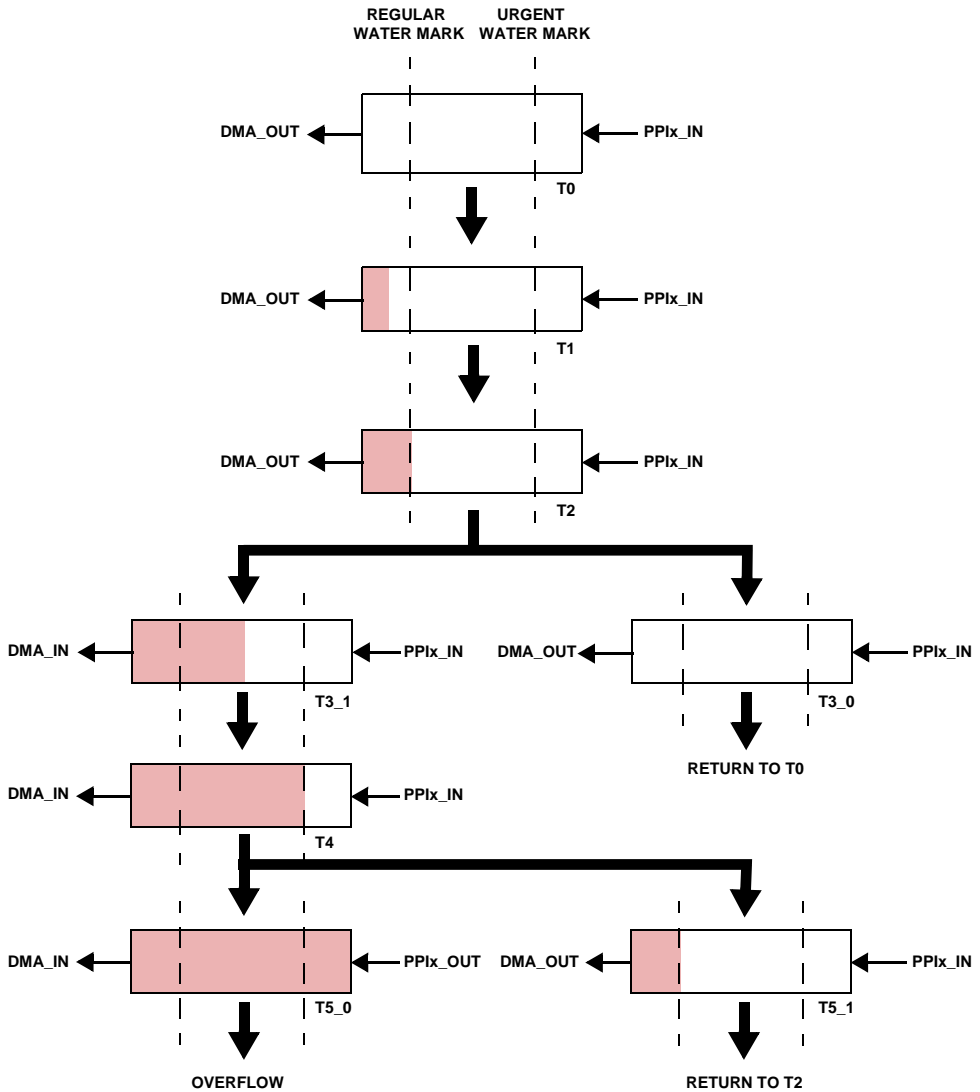


Figure 26-10. FIFO States during Receive Modes

Programming Model

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, the `CORE_EPPI_PPIO` and `SYS_EPPI_PPIO` bit in the `HMDMA1_CONTROL` register (reference the register in chapter here) help to further elevate the priority of EPPI (EPPI0/1/2) transactions at the DDR controller interface. Use of this bit is required only under high DDR activity resulting from core and several other DMA channels (including EPPI-DMA channels) simultaneously accessing the DDR memory.

Due to the pipelined nature of the requests at the DDR interface and the DEB bus submitting a DDR request at a maximum of every other `SCLK` cycle, lower priority resources can gain access of the DDR interface. These two bits are provided to ensure that under EPPI urgent conditions, only EPPI can gain access to the DDR memory, resulting in efficient use of the external memory bus bandwidth.

Enhanced Parallel Peripheral Interface

Setting `CORE_EPPI_PPIO` bit in the `HMDMA1_CONTROL` register, will block all core accesses to the DDR memory as long as any EPPI's request stays urgent or for a maximum period of 124 system clock cycles. After a period of 124 system clock cycles, the core can gain access for a period of 4 system clock cycles.

Setting `SYS_EPPI_PPIO` bit in the `HMDMA1_CONTROL` register will block all DMA channels in DMAC0, as well as USB, PIXC and DMAC1 MDMA channels as long as any EPPI's 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 each EPPI (0, 1 or 2) by enabling the urgent watermark in the respective EPPI control register.

As an exception, a TESTSET instruction has higher priority than EPPI-DMA urgent requests under all conditions. So even when `CORE_EPPI_PPIO` bit of the `HMDMA1_CONTROL` register is set, a TESTSET instruction will not be blocked under 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 one observes EPPI urgent conditions and if using the DMA in descriptor mode, it is recommended to place the DMA descriptors in L2 or L1 memory.

For more information, please refer to section "Handshake MDMA Control (HMDMA_x_CONTROL) Registers" in *ADSP-BF54x Blackfin Hardware Reference Volume 1 of 2*.

System Configuration

Due to pin muxing, there are restrictions on the possible system configurations of the EPPI channels. [Table 26-41](#) shows the possible system configurations.

Table 26-41. EPPI System Configurations

EPPI 0	EPPI 1	EPPI 2
8-24 bits	<i>Not supported</i>	<i>Not supported</i>
8-18 bits	8 bits	8 bits
8-18 bits	10-14 bits	<i>Not supported</i>
8-18 bits	8 bits	<i>Not supported</i>
8-18 bits	16 bits	<i>Not supported</i>
8-24 bits	<i>Not supported</i>	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 26-42](#) contains a list of EPPI memory-mapped registers (MMRs). Default values of all MMRs are 0x0, except PPIX_CLIP whose default value is 0xFF00FF00.

Table 26-42. EPPI Memory-Mapped Registers

Symbol	Name	Width	Address
PPIX_STATUS	EPPI status registers on page 26-86	16	0xFFC01000
PPIX_HCOUNT	EPPI horizontal transfer count registers on page 26-95	16	0xFFC01004

Enhanced Parallel Peripheral Interface

Table 26-42. EPPI Memory-Mapped Registers (Cont'd)

Symbol	Name	Width	Address
PPIx_HDELAY	EPPI horizontal delay registers on page 26-94	16	0xFFC01008
PPIx_VCOUNT	EPPI vertical transfer count registers on page 26-93	16	0xFFC0100C
PPIx_VDELAY	EPPI vertical delay registers on page 26-93	16	0xFFC01010
PPIx_FRAME	EPPI lines per frame registers on page 26-92	16	0xFFC01014
PPIx_LINE	EPPI samples per line registers on page 26-92	16	0xFFC01018
PPIx_CLKDIV	EPPI clock divide registers on page 26-95	16	0xFFC0101C
PPIx_CONTROL	EPPI control registers on page 26-79	32	0xFFC01020
PPIx_FS1W_HBL	EPPI FS1 width register/ EPPI horizontal blanking samples per line registers on page 26-96	32	0xFFC01024
PPIx_FS1P_AVPL	EPPI FS1 period register/ EPPI active video samples per line registers on page 26-98	32	0xFFC01028
PPIx_FS2W_LVB	EPPI FS2 width register/ EPPI lines of vertical blanking registers on page 26-96	32	0xFFC0102C
PPIx_FS2P_LAVF	EPPI FS2 period register/ EPPI lines of active video per field registers on page 26-99	32	0xFFC01030
PPIx_CLIP	EPPI clipping registers on page 26-101	32	0xFFC01034

The MMR addresses shown in [Table 26-42](#) refer to the PPIx registers, that start at a base address of 0xFFC01000. PPI1 and PPI2 have base addresses of 0xFFC01300 and 0xFFC02900, respectively, and follow the same register address increments as shown above for PPIx.

EPPI Registers

Table 26-43 shows which of the MMRs are valid for which operating modes (an “X” indicates that the register is valid for the particular mode):

Table 26-43. 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	
PPIx_FRAME					X	X	X	X	X	X	X	X					X
PPIx_LINE	X	X	X	X	X	X	X	X	X	X	X	X					X
PPIx_HDELAY	X	X	X	X	X	X	X	X	X	X	X	X					
PPIx_HCOUNT	X	X	X	X	X	X	X	X	X	X	X	X					
PPIx_VDELAY					X	X	X	X	X	X	X	X					
PPIx_VCOUNT						X	X	X	X	X	X	X					
PPIx_FS1W_HBL			X	X			X	X			X	X					
PPIx_FS1P_AVPL			X	X			X	X			X	X					
PPIx_FS2W_LVB							X	X			X	X					
PPIx_FS2P_LAVF							X	X			X	X					

PPIx_CLKDIV is valid for all modes when an internal clock is used (ICLKEN = 1 in PPIx_CONTROL).

PPIx_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 PPIx_FS1/PPIx_FS2. In GP 0 FS transmit mode when BLANKGEN = 1, they are used as internal blanking generation registers:

PPIX_FS2_WIDTH = PPIX_LVB (Lines of Vertical Blanking)

PPIX_FS2_PERIOD = PPIX_LAVF (Lines of Active Video per Field)


PPIX_FS1_WIDTH = PPIX_HBL (Horizontal Blank Samples per Line)

PPIX_FS1_PERIOD = PPIX_AVPL (Active Video Samples per Line)


Each pair of the above registers has the same physical address.

PPIX_CONTROL Register

The PPIX_CONTROL register, shown in [Figure 26-11](#) and [Figure 26-12](#), is a 32-bit register that configures the EPPI for operating mode, control signal polarities, and data width of the port.

 Please be aware that [Figure 26-11](#) and [Figure 26-12](#) split the PPIX_CONTROL register “halves” across nonstandard boundaries in order to maintain the DLEN field intact.

The PPIX_EN when set, enables the EPPI for operation. On disabling the EPPI by writing a 0 to the EN bit of PPIX_CONTROL, all EPPI MMRs, except PPIX_STATUS, do not return to their reset values. EPPI Interrupt and DMA requests go inactive. Internally generated PPIX_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.

EPPI Registers

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 `PPIX_FS1`. A value of `b#10` means that `HSYNC` is driven on `PPIX_FS1` and that `VSYNC` is driven on `PPIX_FS2`. A value of `b#11` means that `HSYNC` is driven on `PPIX_FS1`, that `VSYNC` is driven on `PPIX_FS2`, and that `FIELD` is driven on `PPIX_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 `PPIX_FS3` is toggled on every assertion of `PPIX_FS2` (if set, =1) or if the `PPIX_FS3` is toggled on every `PPIX_FS1` assertion followed by `PPIX_FS2` assertion (if cleared, =0)

Enhanced Parallel Peripheral Interface

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. **BLANKGEN** 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 **PPIx_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 & 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 26-41 on page 26-76](#) shows the possible configurations. In ITU-R 656 modes, the **DLEN** field should be configured for 8- or 10-bit width.

EPPI Registers

PPIx Control Register (PPIx_CONTROL), Lower Half

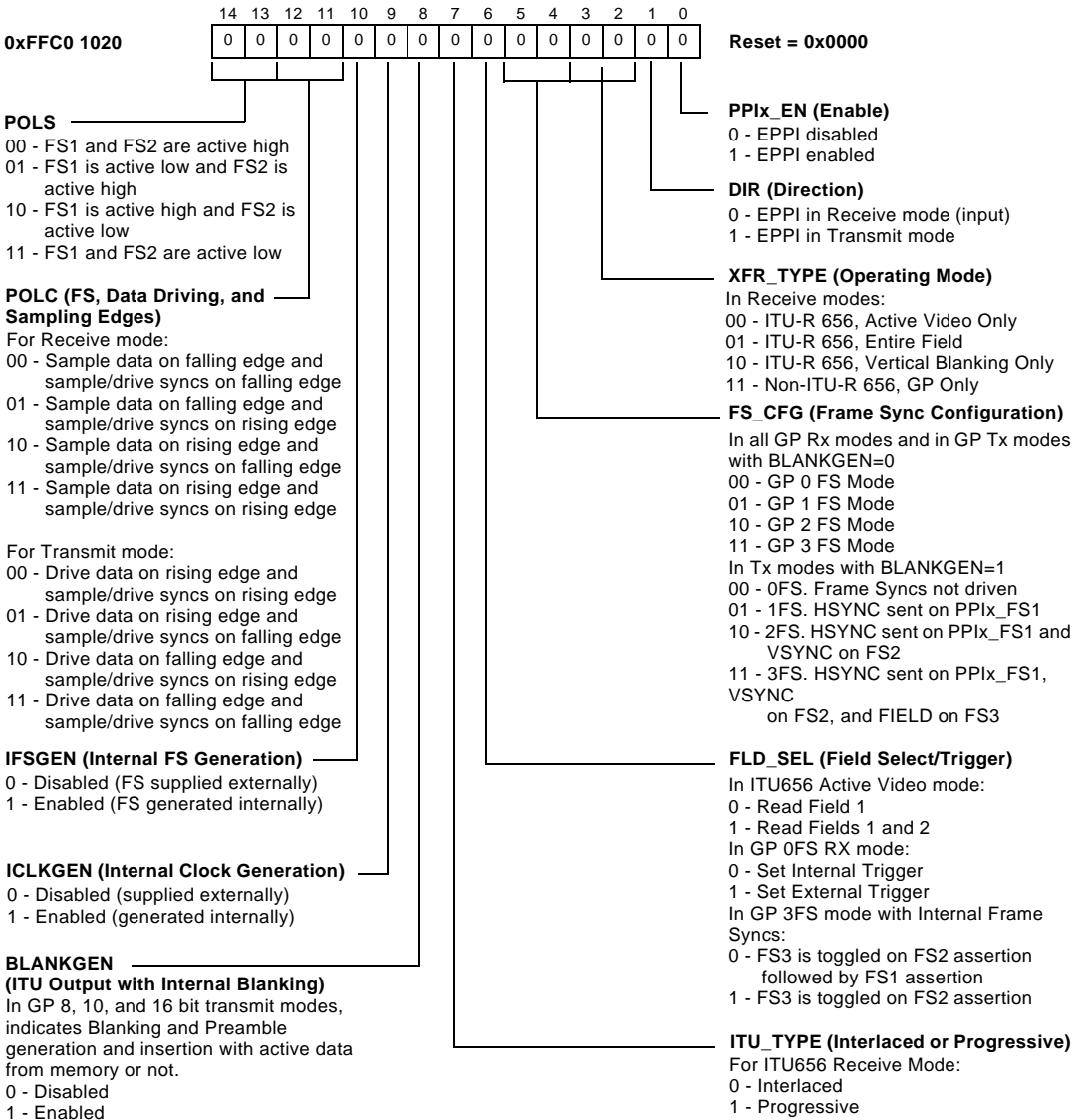


Figure 26-11. PPIx Control Register, Lower Half

Enhanced Parallel Peripheral Interface

PPIx Control Register (PPIx_CONTROL) Upper Half

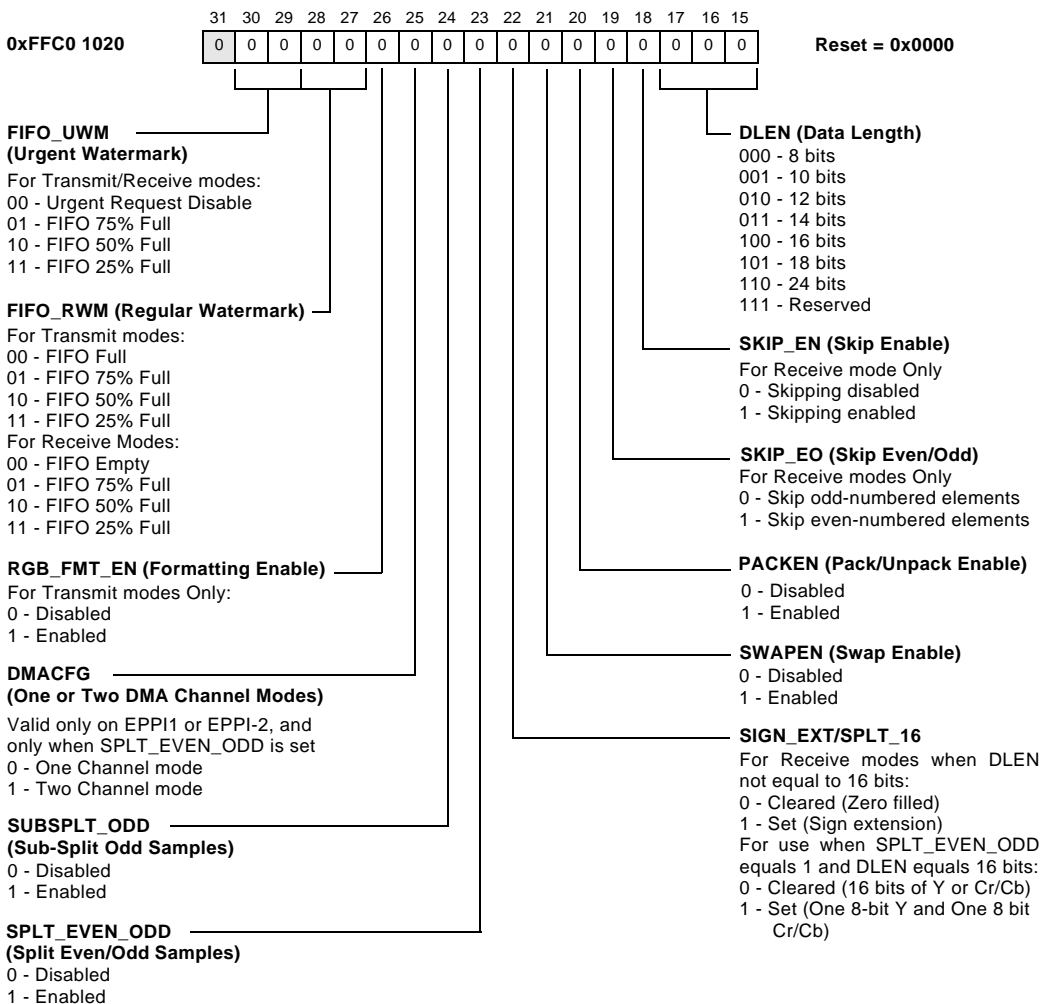


Figure 26-12. PPIx Control Register, Upper Half

EPPI Registers

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 26-33](#) and [“Split Transmit Modes” on page 26-33](#) 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.

EPPI Registers

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.

PPIx_STATUS Register

The `PPIX_STATUS` register, shown in [Figure 26-13](#), is a 16-bit register that indicates the status of the EPPI.


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 PPIx_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.

EPPI Registers

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).

Enhanced Parallel Peripheral Interface

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)

EPPI Registers

PPIx Status Register (PPIx_STATUS)

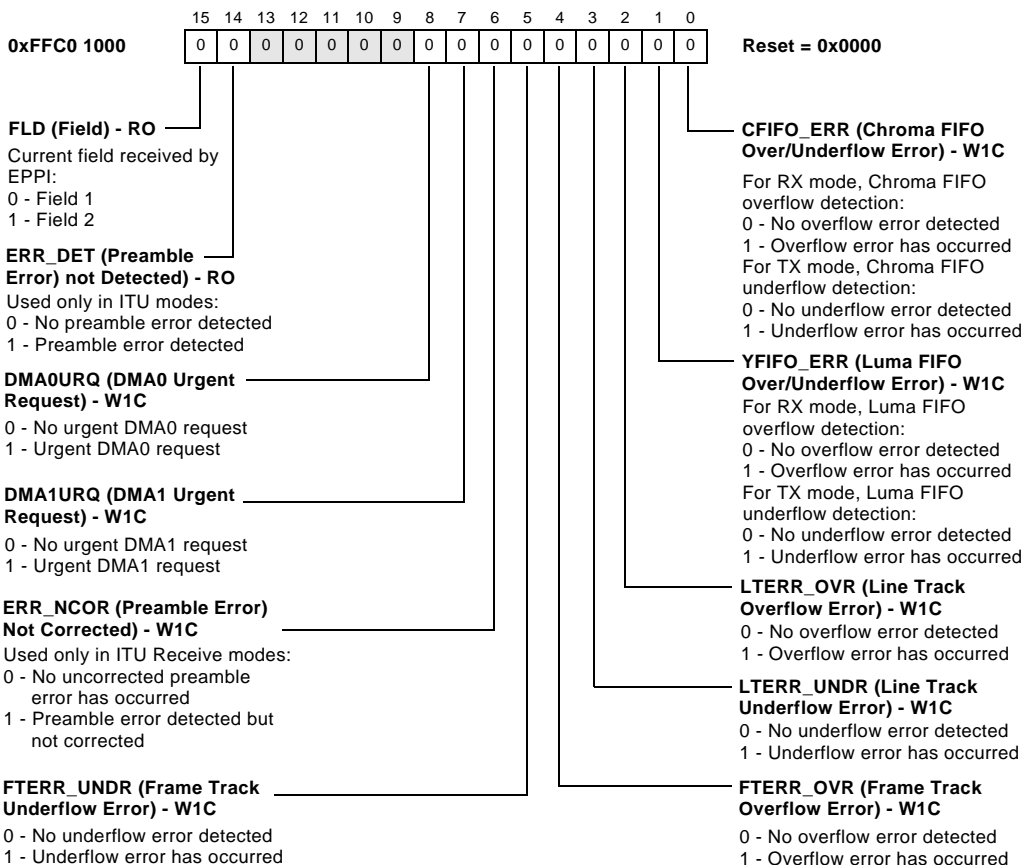


Figure 26-13. EPPI Status Register

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 reduc-

tion. 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 26-14](#).

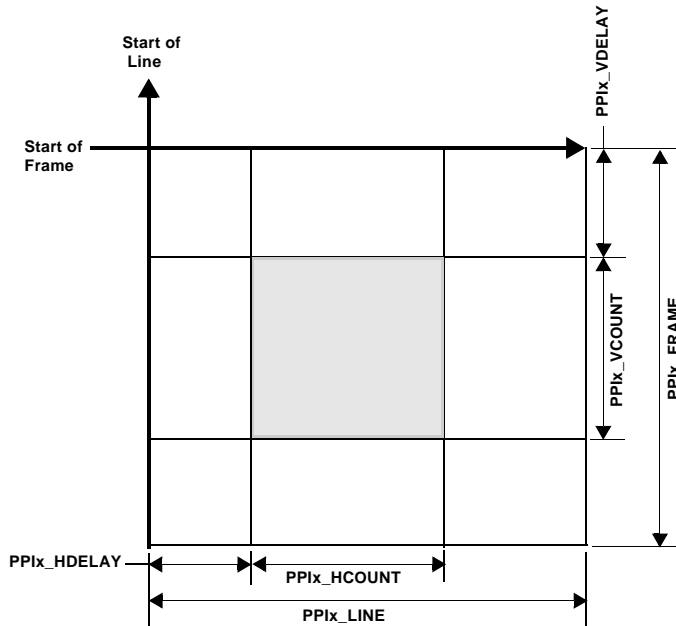


Figure 26-14. Windowing Registers to define a Frame

The shaded portion in [Figure 26-14](#) 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:

$$PPIx_VDELAY + PPIx_VCOUNT \leq PPIx_FRAME$$

$$PPIx_HDELAY + PPIx_HCOUNT \leq PPIx_LINE$$

EPPI Registers

EPPI Lines per Frame Register (PPIx_FRAME)

The PPIx_FRAME register, shown in [Figure 26-15](#), 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 PPIx_FRAME register will also write the same value to the PPIx_VCOUNT register. However, any write to PPIx_VCOUNT does not affect the PPIx_FRAME register value. Therefore, the PPIx_FRAME register should be programmed before the PPIx_VCOUNT register.

Lines per Frame Register (PPIx_FRAME)

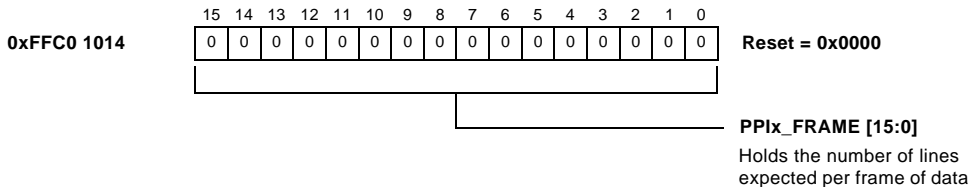


Figure 26-15. EPPI Lines per Frame Register

EPPI Samples per Line Register (PPIx_LINE)

The PPIx_LINE register, shown in [Figure 26-16](#), 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 PPIx_LINE register will also write the same value to the PPIx_HCOUNT register. However, any write to PPIx_HCOUNT does not affect the PPIx_LINE register value. Therefore, the PPIx_LINE register should be programmed before the PPIx_HCOUNT register.

Samples per Line Register (PPIx_LINE)

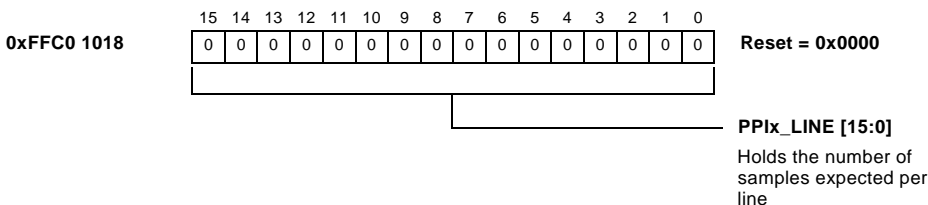


Figure 26-16. EPPI Samples per Line Register

EPPI Vertical Delay Register (PPIx_VDELAY)

The PPIx_VDELAY register, shown in Figure 26-17, 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.

Vertical Delay Count Register (PPIx_VDELAY)

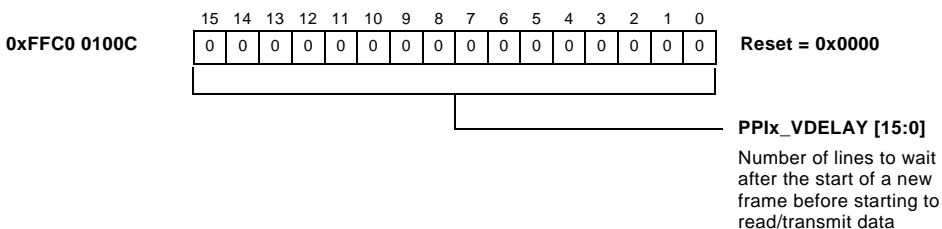


Figure 26-17. EPPI Vertical Delay Count Register

EPPI Vertical Transfer Count Register (PPIx_VCOUNT)

The PPIx_VCOUNT register, shown in Figure 26-18, is a 16-bit register and holds the number of lines to read in or write out, after PPIx_VDELAY number of lines from the start of frame. Any write to the PPIx_FRAME register

EPPI Registers

modifies the `PPIX_VCOUNT` register. However, any write to `PPIX_VCOUNT` does not affect the `PPIX_FRAME` register value. Therefore, the `EPPIO_VCOUNT` register should be programmed after the `PPIX_FRAME` register.

Vertical Transfer Count Register (`PPIX_VCOUNT`)

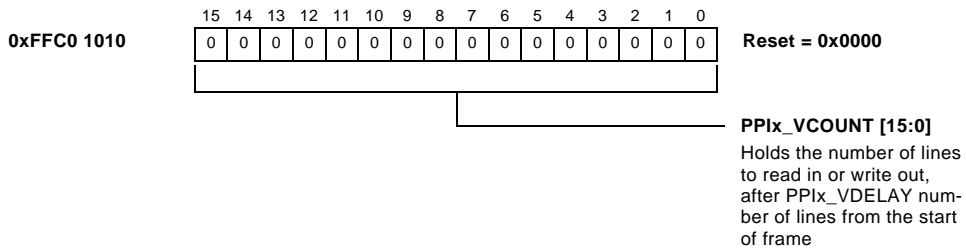


Figure 26-18. EPPI Vertical Transfer Count Register

EPPI Horizontal Delay Register (`PPIX_HDELAY`)

The `PPIX_HDELAY` register, shown in [Figure 26-19](#), is a 16-bit register and contains the number of clock cycles to delay after the assertion of `PPIX_FS1` is detected before starting to read or write data.

Horizontal Delay Register (`PPIX_HDELAY`)

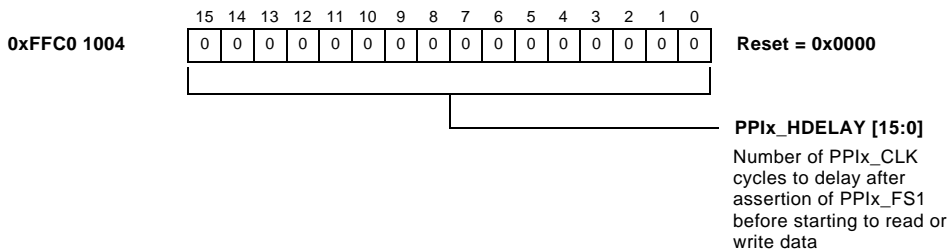


Figure 26-19. EPPI Horizontal Delay Register

EPPi Horizontal Transfer Count Register (PPIx_HCOUNT)

The PPIx_HCOUNT register, shown in [Figure 26-20](#), is a 16-bit register and holds the number of samples to read in or write out per line, after PPIx_HDELAY number of cycles have expired since the assertion of PPIx_FS1. Any write to the PPIx_LINE register modifies the PPIx_HCOUNT register. However, any write to PPIx_HCOUNT does not affect the PPIx_LINE register value. Therefore, the PPIx_HCOUNT register should be programmed after the PPIx_LINE register.

Horizontal Transfer Count Register (PPIx_HCOUNT)

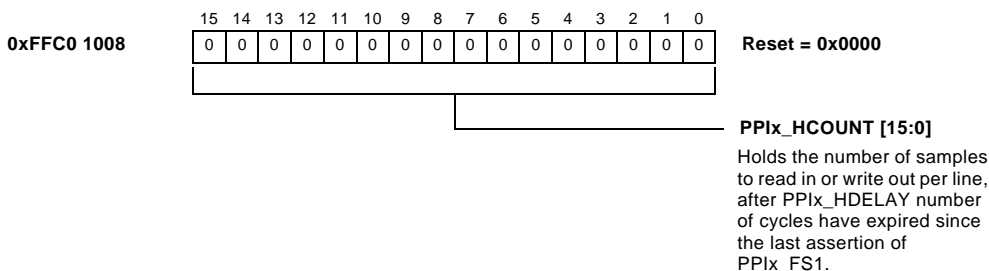


Figure 26-20. EPPi Horizontal Transfer Count Register

EPPi Clock Divide Register (PPIx_CLKDIV)

The PPIx_CLKDIV register, shown in [Figure 26-21](#), is a 16-bit register used for internal clock generation. The generated clock frequency is given by following formula:

$$PPIx_CLK = (SCLK) / (2 * (PPIx_CLKDIV[15:0] + 1))$$

EPPI Registers

Note that a value of 0xFFFF is invalid for PPIx_CLKDIV register.

Clock Divide Register (PPIx_CLK)

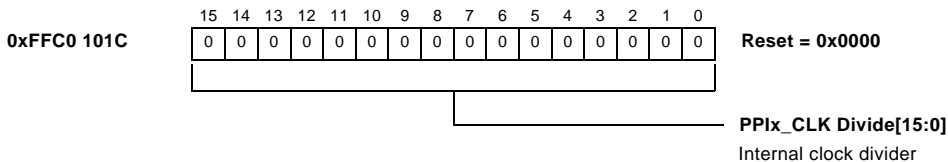


Figure 26-21. EPPI Clock Divide Register (PPIx_CLK)

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 (PPIx_FS1W_HBL)

The PPIx_FS1W_HBL register, shown in [Figure 26-22](#), 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 PPIx_CLK.

In GP Transmit mode with BLANKGEN = 1 in PPIx_CONTROL, it contains the number of samples of horizontal blanking per line.

When used for blanking generation, only the lower 16 bits are valid.

i A value of 0 for this register is illegal. If it is programmed as 0, the EPPI will regard its value as 1.

EPPI FS2 Width Register/ EPPI Lines of Vertical Blanking Register (PPIx_FS2W_LVB)

PPIx_FS2W_LVB, shown in [Figure 26-23](#), is a 32-bit register.

Enhanced Parallel Peripheral Interface

PPIx_FS1 Width / Horizontal Blanking Samples per Line Register (PPIx_FS1W_HBL)

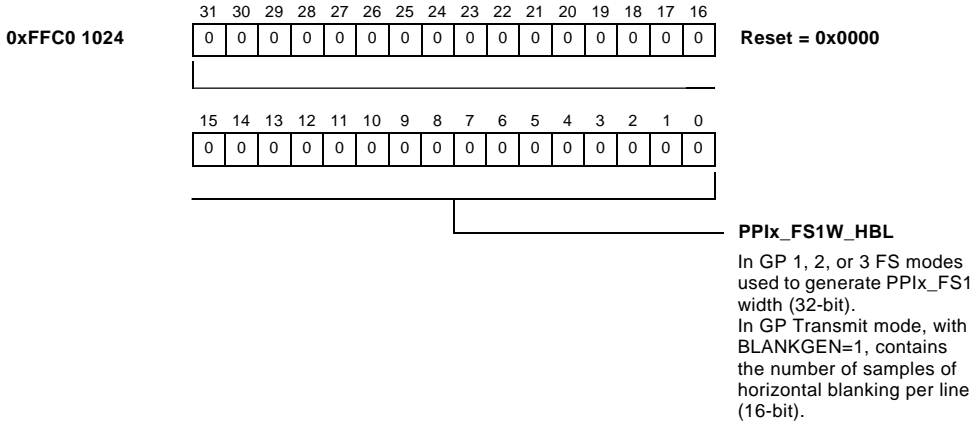


Figure 26-22. EPPI FS1 Width / Horizontal Blanking Samples per Line Register

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 PPIx_CLK.

EPPI Registers

In GP Transmit mode with `BLANKGEN=1` in `PPIx_CONTROL`, it contains the number or lines of vertical blanking.

FS2 Width Register/EPPI Lines of Vertical Blanking Register (EPPIO_FS2W_LVB)

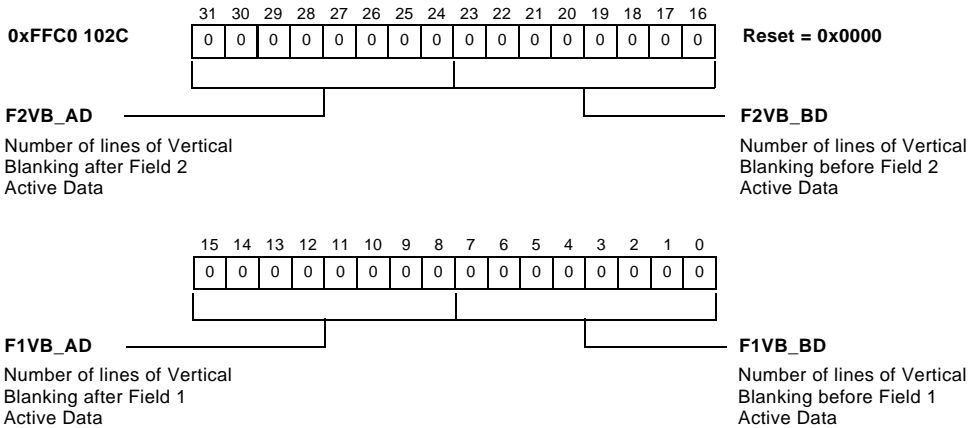


Figure 26-23. EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register

For progressive video, `F2VB_BD` and `F2VB_AD` are ignored.

EPPI FS1 Period Register/EPPI Active Video Samples per Line Register (PPIx_FS1P_AVPL)

The `PPIx_FS1P_AVPL` register, shown in [Figure 26-24](#), 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 `PPIx_FS1`. The reference clock is `PPIx_CLK`.

In GP Transmit mode with `BLANKGEN = 1` in `PPIX_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.

i A value of 0 for this register is illegal. If it is programmed as 0, the EPPI will regard its value as 1.

PPIX_FS1 Period Register / EPPI Active Video Samples per Line Register (PPIX_FS1P_AVPL)

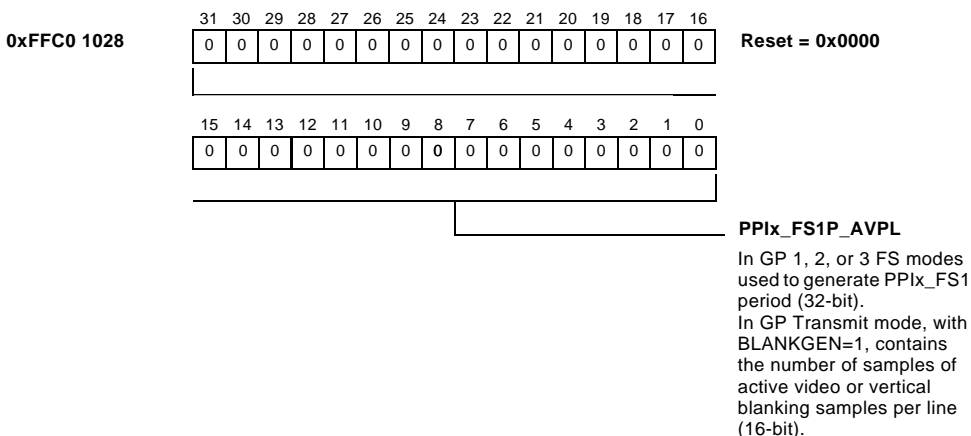


Figure 26-24. EPPI FS1 Period Register / EPPI Active Video Samples per Line Register (PPIX_FS1P_AVPL)

EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (PPIX_FS2P_LAVF)

The `PPIX_FS2_PERIOD` register, shown in [Figure 26-25](#), 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 `PPIX_CLK`.

EPPI Registers

In GP Transmit mode with `BLANKGEN=1` in `PPIx_CONTROL`, it contains the number of lines of active video per field.

FS2 Period Register / EPPI Lines of Active Video per Frame Register (`PPIx_FS2_LVF`)

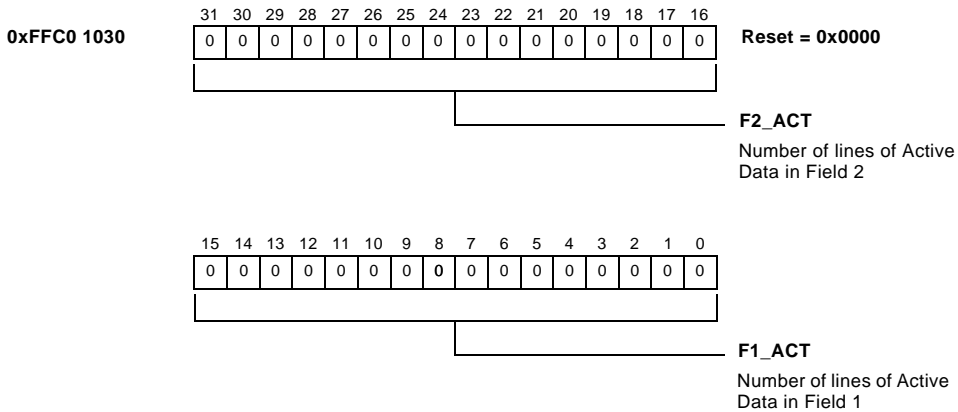


Figure 26-25. EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (`PPIx_FS2_LAVF`)

For progressive video, `F2_ACT` is ignored.

i 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 (PPIx_CLIP)

The PPIx_CLIP register, shown in Figure 26-26, 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 26-26 for bit definitions.

Clipping Register (PPIx_CLIP)

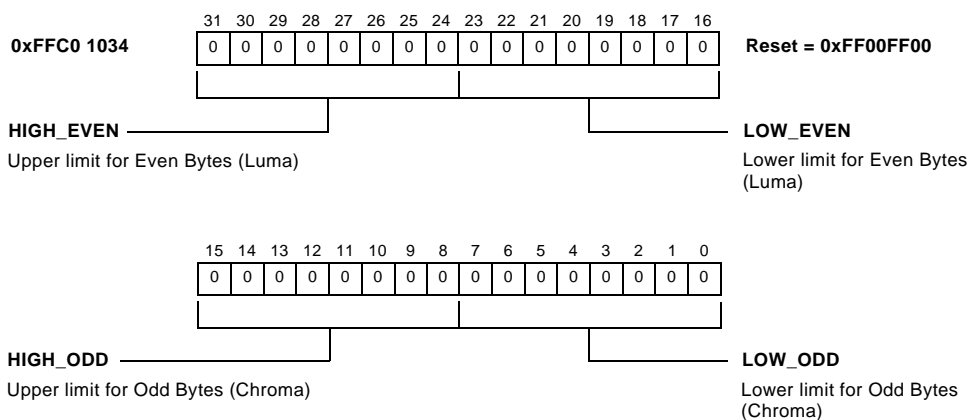



Figure 26-26. EPPI Clipping Register (PPIx_CLIP)

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.

EPPI Registers

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.

 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.

Enhanced Parallel Peripheral Interface

27 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 27-1
- “Interface Overview” on page 27-2
- “CAN Operation” on page 27-10
- “Functional Operation” on page 27-25
- “CAN Registers” on page 27-41
- “Programming Examples” on page 27-91

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. Please see *ADSP-BF54x Blackfin Embedded Processor* datasheet for more information.

Interface Overview

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 27-1](#) for a symbolic representation of the CAN transceiver interconnection, and [Figure 27-2](#) 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.

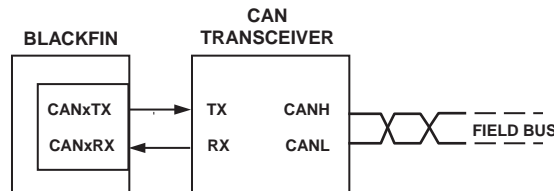


Figure 27-1. Representation of CAN Transceiver Interconnection

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.

Interface Overview

BLACKFIN

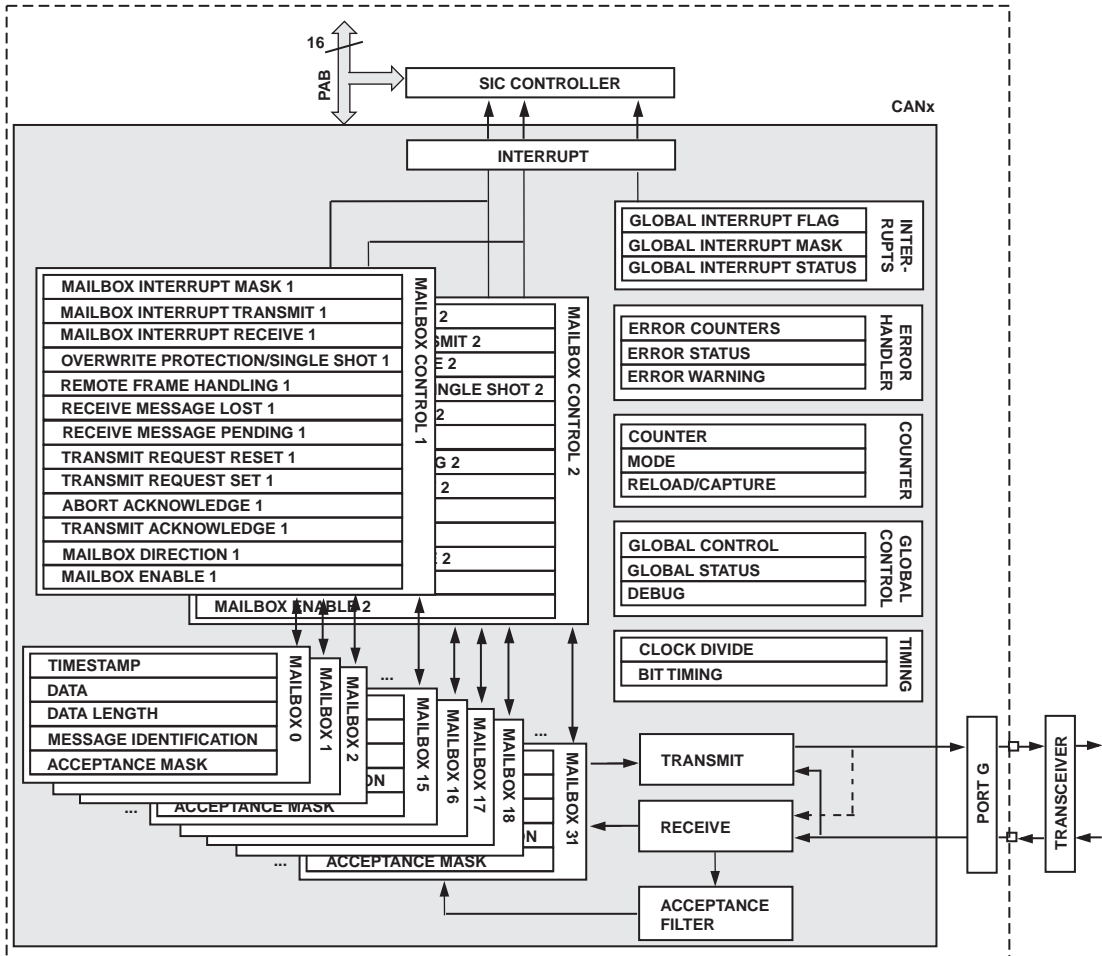


Figure 27-2. CANx Block Diagram

The PG13 pin (CANORX 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 27-3](#).

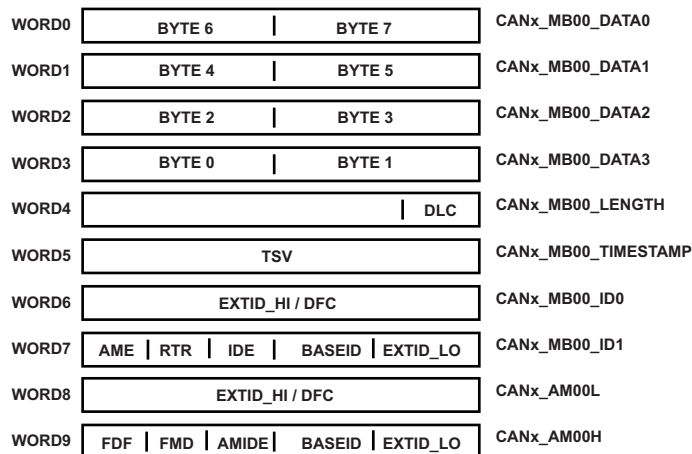


Figure 27-3. CAN Mailbox Area

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

Interface Overview

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_ID0/1`) register pair includes:

- The 29 bit identifier (base part `BASEID` plus extended part `EXTID_LO/HI`)
- The acceptance mask enable bit (`AME`)
- The remote transmission request bit (`RTR`)
- The identifier extension bit (`IDE`)



Do not write to the identifier of a message object while the mailbox is enabled for the CAN module (the corresponding bit in `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 CAN_x_CONTROL register and FDF = 1 in the corresponding acceptance mask), the EXTID_HI[15:0] bits of CAN_x_MB_{xx}_ID0 are reused as acceptance code (DFC) for the data field filtering. For more details, see “Receive Operation” on page 27-16 of this chapter.

CAN Mailbox Control

Mailbox control MMRs function as control and status registers for the 32 mailboxes. Each bit in these registers represents one specific mailbox. Since CAN MMRs are all 16 bits wide, pairs of registers are required to manage certain functionality for all 32 individual mailboxes. Mailboxes 0-15 are configured/monitored in registers with a suffix of 1. Similarly, mailboxes 16-31 use the same named register with a suffix of 2. For example, the CAN mailbox direction registers (CAN_x_MD_x) would control mailboxes as shown in Figure 27-4.

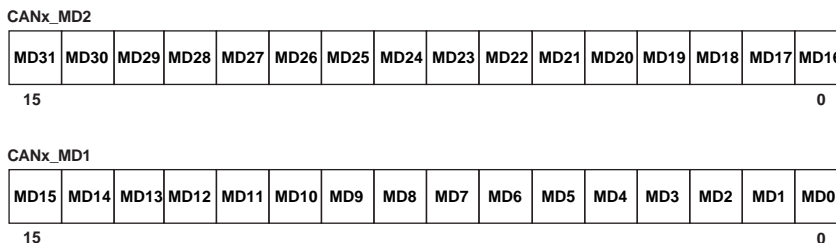


Figure 27-4. CAN Register Pairs

The mailbox control register area consists of these register pairs:

- CAN_x_MC1 and CAN_x_MC2 (mailbox enable registers)
- CAN_x_MD1 and CAN_x_MD2 (mailbox direction registers)

Interface Overview

- 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 27-1 on page 27-3](#)) 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 CAN_{TX} 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 CAN_{RX} pin to verify that the logic level being driven is the value it just placed on the CAN_{TX} pin. This is where the names for the logic levels apply. If a transmitting node places a recessive ‘1’ on CAN_{TX} and detects a dominant ‘0’ on the CAN_{RX} 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 CAN_{RX} is the value driven on CAN_{TX}, 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 27-5](#) for more details regarding CAN frame structure.

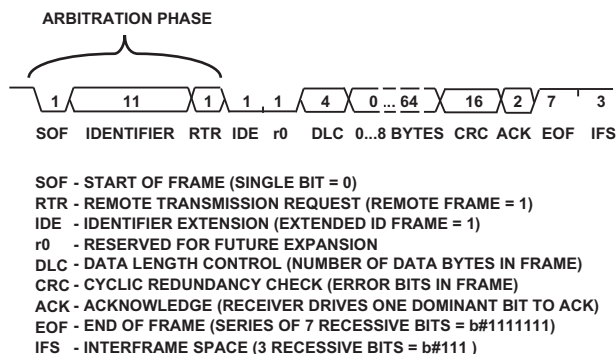


Figure 27-5. Standard CAN Frame

CAN Operation

Figure 27-5 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).

i 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 27-6.

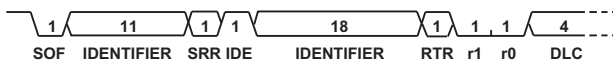


Figure 27-6. Extended CAN Frame

As could be concluded with regards to the *RTR* field, a dominant bit in the *IDE* field wins arbitration against an extended frame with the same lower 11-bits, therefore, standard frames are higher priority than extended frames. The substitute remote request bit (*SRR*, always sent as recessive), the reserved bits *r0* and *r1* (always sent as dominant), and the checksum (*CRC*) are generated automatically by the internal logic.

CAN Operation

The CAN controller is in configuration mode when coming out of processor reset or hibernate. It is only when the CAN is in configuration mode that hardware behavior can be altered. Before initializing the mailboxes themselves, the CAN bit timing must be set up to work on the CAN bus that the controller is expected to connect to.

Bit Timing

The CAN controller does not have a dedicated clock. Instead, the CAN clock is derived from the system clock (SCLK) based on a configurable number of time quanta. The Time Quantum (TQ) is derived from the formula $TQ = (BRP+1)/SCLK$, where BRP is the 10-bit BRP field in the CAN_x_CLOCK register. Although the BRP field can be set to any value, it is recommended that the value be greater than or equal to 4, as restrictions apply to the bit timing configuration when BRP is less than 4.

The CAN_x_CLOCK register defines the TQ value, and multiple time quanta make up the duration of a CAN bit on the bus. The CAN_x_TIMING register controls the nominal bit time and the sample point of the individual bits in the CAN protocol. Figure 27-7 shows the three phases of a CAN bit—the synchronization segment, the segment before the sample point, and the segment after the sample point.

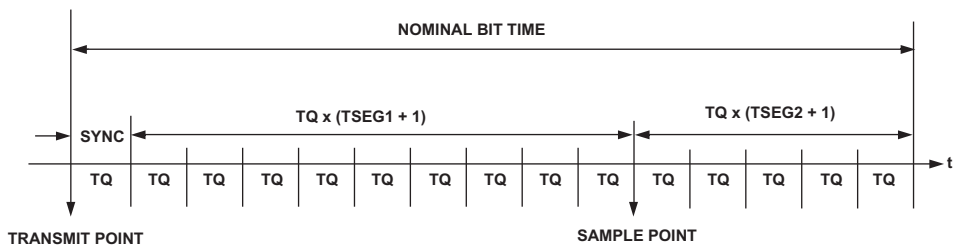


Figure 27-7. Three Phases of a CAN Bit

The synchronization segment is fixed to one TQ. It is required to synchronize the nodes on the bus. All signal edges are expected to occur within this segment.

The TSEG1 and TSEG2 fields of CAN_x_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 $t_{BIT} = TQ \times (1 + (1 + TSEG1) + (1 + TSEG2))$. For safe receive operation on given physical networks, the sample point is pro-

CAN Operation


grammable 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:

$$SJW \leq TSEG2 \leq TSEG1$$


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, how-

ever, 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 `TSEG1` field of the `CANx_TIMING` register is programmed to '0,' the module doesn't leave the configuration mode.

During configuration mode, the module is not active on the CAN bus line. The `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 software 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 software reset (`SRS = 1` in `CANx_CONTROL`).

Transmit Operation

[Figure 27-8](#) 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`).

CAN Operation

When a transmission completes, the corresponding bits in the transmit request set register and in the transmit request reset register (TRR_n 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 TRR_n 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 TRS_n 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 ($OPSS_n = 0$ 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 ($RFH_n = 1$ in $CANx_RFHx$).

Special care should be given to mailbox area management when a TRS_n bit is set. Write access to the mailbox is permissible with TRS_n 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 TRS_n bit associated with a disabled mailbox may result in erroneous behavior. Similarly, disabling a mailbox before the associated TRS_n 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 27-8](#)). The currently aborted transmission is restarted after any messages with higher priority are sent.

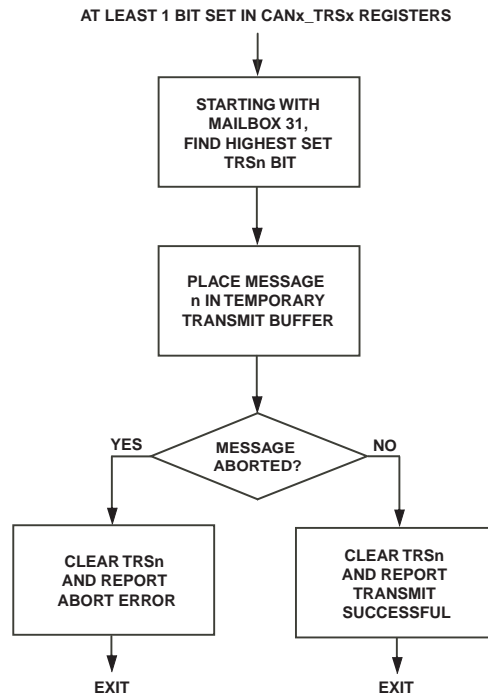


Figure 27-8. CAN Transmit Operation Flow Chart

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message in the buffer is not replaced until it is sent successfully, the arbitration on the CAN bus line is lost, or there is an error frame on the CAN bus line.

Single Shot Transmission

If the single shot transmission feature is used ($OPSS_n = 1$ 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

CAN Operation

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 ($AA_n = 1$ 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 ($MD_n = 1$ and $MC_n = 1$) and to all active transmit mailboxes with the remote frame handling feature enabled ($RFH_n = 1$ 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 27-9](#).

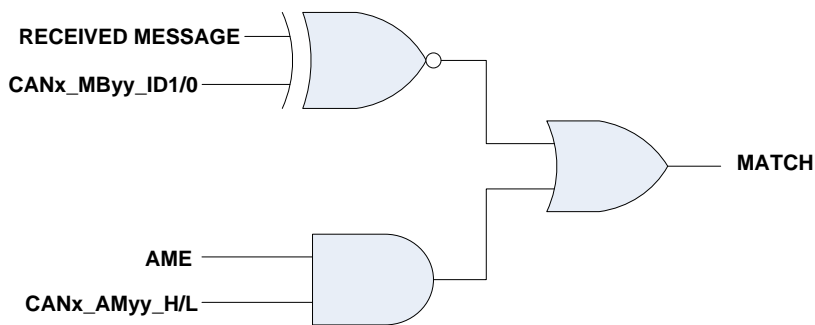


Figure 27-9. CAN Receive Message Logic

CAN Operation

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 27-1. 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 27-10 illustrates the decision tree of the receive logic when processing the individual mailboxes.

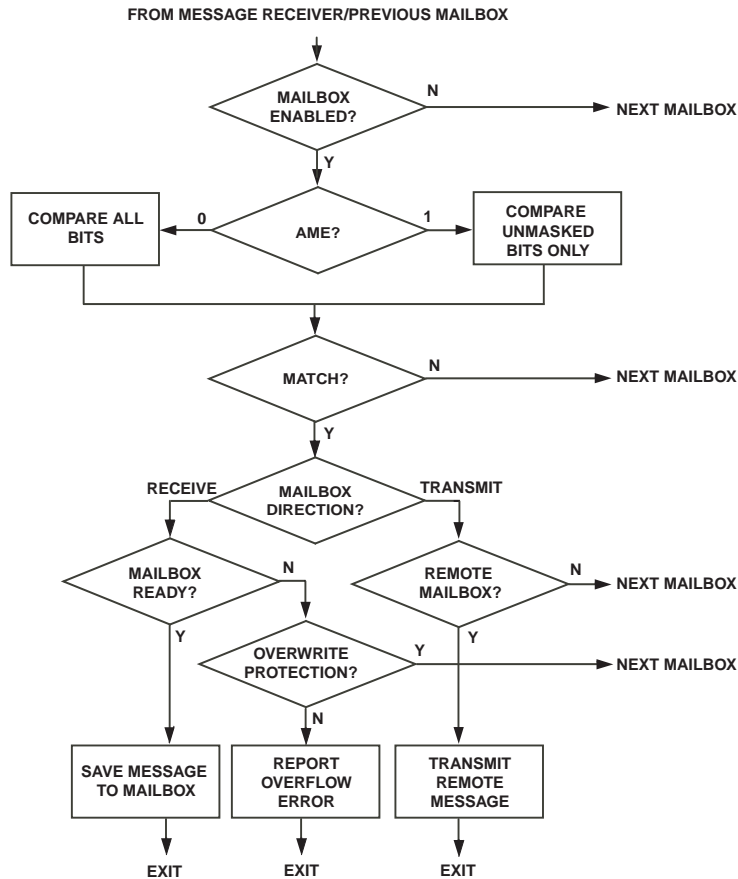



Figure 27-10. CAN Receive Operation Flow Chart

If a message is received for a mailbox and that mailbox still contains unread data ($RMP_n = 1$), the user has to decide whether the old message should be overwritten or not. If $OPSS_n = 0$, the receive message lost bit (RML_n in $CAN_x_RML_x$) is set and the stored message is overwritten. This

CAN Operation

results in the receive message lost interrupt being raised in the global CAN interrupt status register (RMLIS = 1 in CAN_x_GIS). If OPSS_n = 1, the next mailboxes are checked for another matching identifier. If no match is found, the message is discarded and the next message is checked.

 If a receive mailbox is disabled, an ongoing receive message for that mailbox is lost even if a second mailbox is configured to receive the same identifier.

Data Acceptance Filter

If DeviceNet mode is enabled (DNM = 1 in CAN_x_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 27-2](#).

Table 27-2. Data Field Filtering

FDF Filter On Data Field	FMD Full Mask Data Field	Description
0	0	Do not allow filtering on the data field
0	1	Not allowed. FMD must be 0 if FDF is 0.
1	0	Filter on first data byte only
1	1	Filter on first two data bytes

If the FDF bit is set in the corresponding CAN_x_AM_{xx}H register, the CAN_x_AM_{xx}L register holds the data field mask (DFM[15:0]). If the FDF bit is cleared in the corresponding CAN_x_AM_{xx}H register, the CAN_x_AM_{xx}L 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.

CAN Operation

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`).

Remote frames are data frames with no data field and the RTR bit set. The data length code of the data frame is equal to the DLC of the corresponding remote frame. A data length code can be programmed with values in the range of 0 to 15, but data length code values greater than 8 are considered as 8. A remote frame contains:

- the identifier bits
- the control field DLC
- the remote transmission request (RTR) bit

Only configurable mailboxes 8–23 can process remote frames, but all 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 inconsis-

CAN Operation

tent 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 (TRS_n) 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 TRR_n 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

Functional Operation

flag is set after the requested data frame is sent from the mailbox. If any $MBTIF_n$ bits are set in $CANx_MBTIF_x$, the $MBTIRQ$ interrupt output is raised in $CANx_INTR$. In order to clear the $MBTIRQ$ interrupt request, all of the set $MBTIF_n$ bits must be cleared by software by writing a 1 to those set bit locations in $CANx_MBTIF_x$. Additionally, software must clear the associated TAn bit or set the associated $TRSn$ bit to clear the interrupt source that asserts the $MBTIF_n$ 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 software reset has occurred.

There are several interrupt events that can activate this GIRQ interrupt:

- **Access denied interrupt** (ADIM, ADIS, ADIF)
At least one access to the mailbox RAM occurred during a data update by internal logic.
- **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.

Functional Operation

- **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.

Functional Operation

- $UCCNF[3:0] = 0xE$ – 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 (RMP_n 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 software reset, the content of this register remains unchanged.

CAN Error Handling

Error management is an integral part of the CAN standard. Five different kinds of bus errors may occur during transmissions:

- **Bit error** – A bit error can be detected by the transmitting node only. Whenever a node is transmitting, it continuously monitors its receive pin ($CANxRX$) 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 CAN_xRX does not equal what is being transmitted on CAN_xTX.

- **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 CAN_x_ESR as well as the error counter register CAN_x_CEC. In addition to the standard errors, the CAN_x_ESR register features a flag that signals when the CAN_xRX 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 27-11](#).

Functional Operation

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.

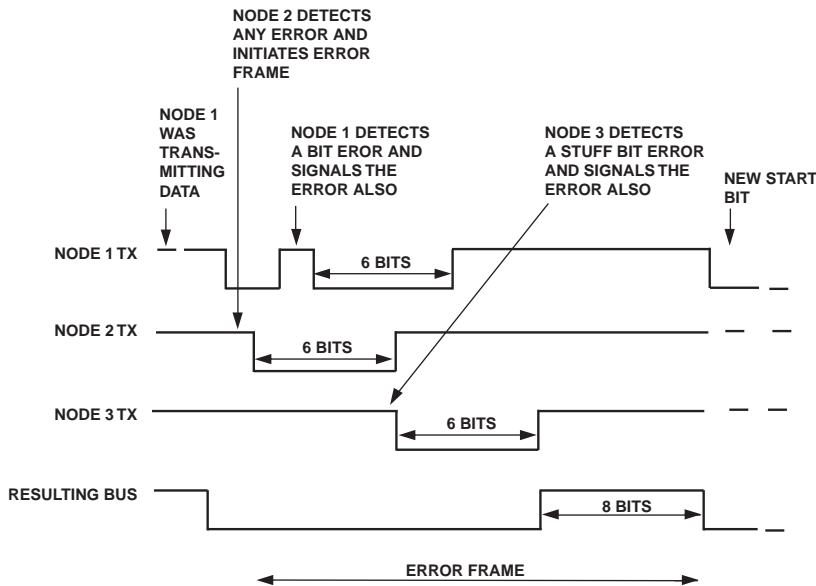


Figure 27-11. CAN Error Scenario Example

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 27-3](#). 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.

Functional Operation

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 27-3. 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 27-90](#).


Additionally, interrupts can occur for all of these levels by unmasking them in the global CAN interrupt mask register (`CANx_GIM`) shown on page 27-50. 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 27-1 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.

Functional Operation

Listing 27-1. 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 27-4 shows several common combinations of test mode bits.

Functional Operation

Table 27-4. 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.
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

Table 27-4. CAN Test Modes (Cont'd)

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
1	1	0	1	1	1	No transmission on CAN bus line. Read back. No external acknowledge required. Neither transmit message nor acknowledge are transmitted on 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.

Functional Operation

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.

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 $50\mu\text{A}$) 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`.

CAN Registers

The following sections describe the CAN controller register definitions.

[Table 27-5](#) through [Table 27-9](#) show the functions of the CAN controller registers.

Table 27-5. CAN Global Registers

Register Name	Function	Notes
<code>CANx_CONTROL</code>	Master control registers on page 27-46	Reserved bits 15:8 and 3 must always be written as ‘0’
<code>CANx_STATUS</code>	Global CAN status registers on page 27-47	Write accesses have no effect
<code>CANx_DEBUG</code>	CAN debug registers on page 27-48	Use of these modes is not CAN-compliant

CAN Registers

Table 27-5. CAN Global Registers (Cont'd)

Register Name	Function	Notes
CANx_CLOCK	CAN clock registers on page 27-48	Accessible only in configuration mode
CANx_TIMING	CAN timing registers on page 27-49	Accessible only in configuration mode
CANx_INTR	CAN interrupt register on page 27-49	Reserved bits 15:8 and 5:4 must always be written as '0'
CANx_GIM	Global CAN interrupt mask registers on page 27-50	Bits 15:11 and 9 are reserved
CANx_GIS	Global CAN interrupt status registers on page 27-51	Bits 15:11 and 9 are reserved
CANx_GIF	Global CAN interrupt flag registers on page 27-52	Bits 15:11 and 9 are reserved

Table 27-6. CAN Mailbox/Mask Registers

Register Name	Function	Notes
CANx_AMxxH CANx_AMxxL	CAN mailbox acceptance registers on page 27-53	Do not write when mailbox MBxx is enabled
CANx_MBxx_ID1 CANx_MBxx_ID0	CAN mailbox word 7 registers CAN mailbox word 6 registers on page 27-57	Do not write when mailbox MBxx is enabled
CANx_MBxx_TIMESTAMP	CAN mailbox word 5 registers on page 27-61	Holds timestamp information when timestamp mode is active
CANx_MBxx_LENGTH	CAN mailbox word 4 registers on page 27-63	Values greater than 8 are not allowed. Bits 15:4 are reserved.
CANx_MBxx_DATA3 CANx_MBxx_DATA2 CANx_MBxx_DATA1 CANx_MBxx_DATA0	CAN mailbox word 3 registers CAN mailbox word 2 registers CAN mailbox word 1 registers CAN mailbox word 0 registers on page 27-65	Software controls reading correct data, based on DLC

Table 27-7. CAN Mailbox Control Registers

Register Name	Function	Notes
CANx_MC1 CANx_MC2	CAN mailbox configuration registers on page 27-73	Always disable before modifying mailbox area or direction
CANx_MD1 CANx_MD2	CAN mailbox direction registers on page 27-74	Never change MDn direction when mailbox n is enabled. MD[31:24] and MD[7:0] are read only
CANx_RMP1 CANx_RMP2	CAN receive message pending registers on page 27-75	Clearing RMPn bits also clears corresponding RMLn bits
CANx_RML1 CANx_RML2	CAN receive message lost registers on page 27-76	Write accesses have no effect
CANx_OPSS1 CANx_OPSS2	CAN overwrite protection or single-shot transmission registers on page 27-77	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	CAN transmission request set registers on page 27-78	May be set by internal logic under certain circumstances. TRS[7:0] are read-only
CANx_TRR1 CANx_TRR2	CAN transmission request reset registers on page 27-79	TRRn bits must not be set if mailbox n is disabled or TRSn = 0
CANx_AA1 CANx_AA2	CAN abort acknowledge registers on page 27-80	AAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CANx_TA1 CANx_TA2	CAN transmission acknowledge registers on page 27-81	TAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CANx_MBTD	CAN temporary mailbox disable registers on page 27-82	Allows safe access to data field of an enabled mailbox

CAN Registers

Table 27-7. CAN Mailbox Control Registers (Cont'd)

Register Name	Function	Notes
CANx_RFH1 CANx_RFH2	CAN remote frame handling registers on page 27-83	Available only to configurable mailboxes 23:8. RFH[31:24] and RFH[7:0] are read-only
CANx_MBIM1 CANx_MBIM2	CAN mailbox interrupt mask registers on page 27-84	Mailbox interrupts are raised only if these bits are set
CANx_MBTIF1 CANx_MBTIF2	CAN mailbox transmit interrupt flag registers on page 27-85	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBTIFn = 1 results in MBRIFn = 1 and MBTIFn = 0
CANx_MBRIF1 CANx_MBRIF2	CAN mailbox receive interrupt flag registers on page 27-86	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBRIFn = 1 results in MBTIFn = 1 and MBRIFn = 0

Table 27-8. CAN Universal Counter Registers

Register Name	Function	Notes
CANx_UCCNF	CAN universal counter mode registers on page 27-87	Bits 15:8 and bit 4 are reserved
CANx_UCCNT	CAN universal counter registers on page 27-88	Counts up or down based on universal counter mode
CANx_UCRC	CAN universal counter reload/capture registers on page 27-88	In timestamp mode, holds time of last successful transmit or receive

Table 27-9. CAN Error Registers

Register Name	Function	Notes
CANx_CEC	CAN error counter registers on page 27-89	Undefined while in bus off mode, not affected by software reset
CANx_ESR	CAN error status registers on page 27-89	Only the first error is stored. SA0 flag is cleared by recessive bit on CAN bus
CANx_EWR	CAN error counter warning level registers on page 27-90	Default is 96 for each counter

CAN Registers

Global CAN Registers

Figure 27-12 through Figure 27-20 on page 27-52 show the CAN global registers.

CANx_CONTROL Master Control Registers

Master Control Register (CANx_CONTROL)

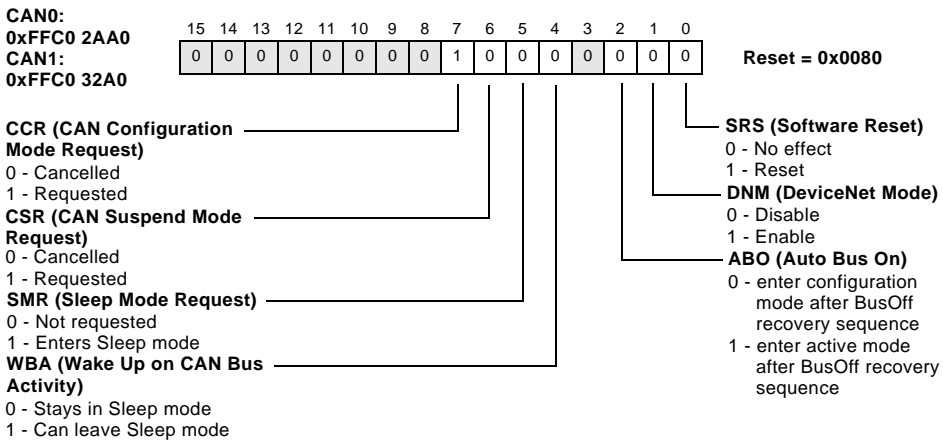


Figure 27-12. Master Control Registers

CANx_STATUS Global CAN Status Registers

Global CAN Status Register (CANx_STATUS)

RO

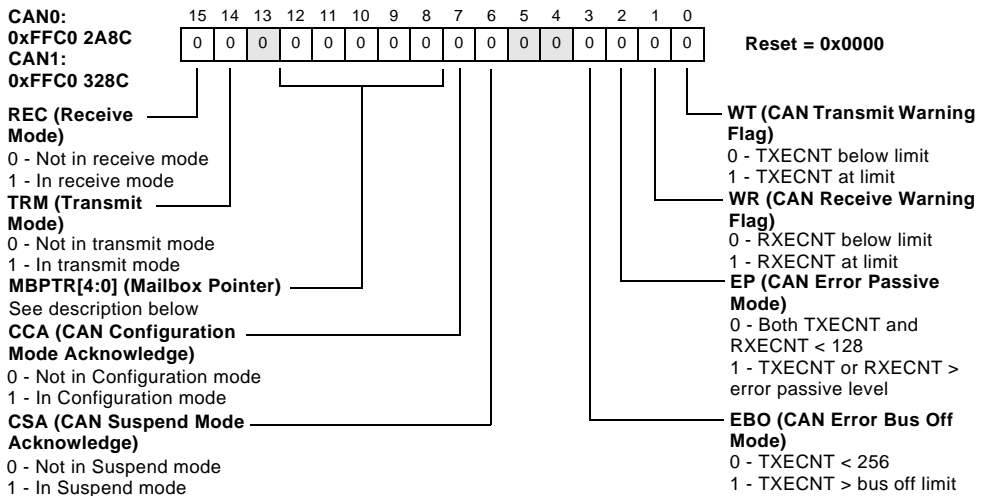


Figure 27-13. Global CAN Status Registers

- **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.

...

...

...

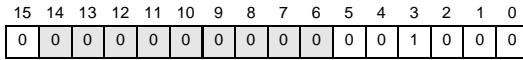
[b#00000] The message of mailbox 0 is currently being processed.

CAN Registers

CANx_DEBUG Registers

CAN Debug Register (CANx_DEBUG)

0xFFC0 2A88



Reset = 0x0008

CDE (CAN Debug Mode Enable)

0 - Debug mode disabled
1 - Debug mode enabled

MRB (Mode Read Back)

0 - Read back mode disabled
1 - Read back mode enabled

MAA (Mode Auto Acknowledge)

0 - Auto acknowledge mode disabled
1 - Auto acknowledge mode enabled

DIL (Disable Internal Loop)

0 - Enable internal loop
1 - Disable internal loop

DEC (Disable Transmit and Receive Error Counters)

0 - Enable CANx_CEC Tx and Rx error counters
1 - Disable CANx_CEC Tx and Rx error counters

DRI (Disable Receive Input Pin, CANxRX)

0 - Enable CANxRX input pin
1 - Disable CANxRX input pin-drive recessive internally

DTO (Disable Transmit Output Pin, CANxTX)

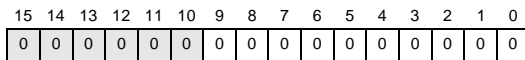
0 - Enable CANxTX output pin
1 - Disable CANxTX output pin-drive recessive

Figure 27-14. CAN Debug Registers

CANx_CLOCK Registers

CAN Clock Register (CANx_CLOCK)

0xFFC0 2A80



Reset = 0x0000

BRP[9:0] (Bit Rate Prescaler Register) W/R

Figure 27-15. CAN Clock Registers

CANx_TIMING Registers

CAN Timing Register (CANx_TIMING)

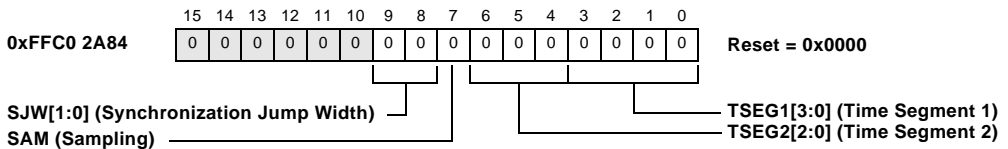


Figure 27-16. CAN Timing Registers

CANx_INTR Interrupt Pending Registers

CAN Interrupt Register (CANx_INTR)

RO

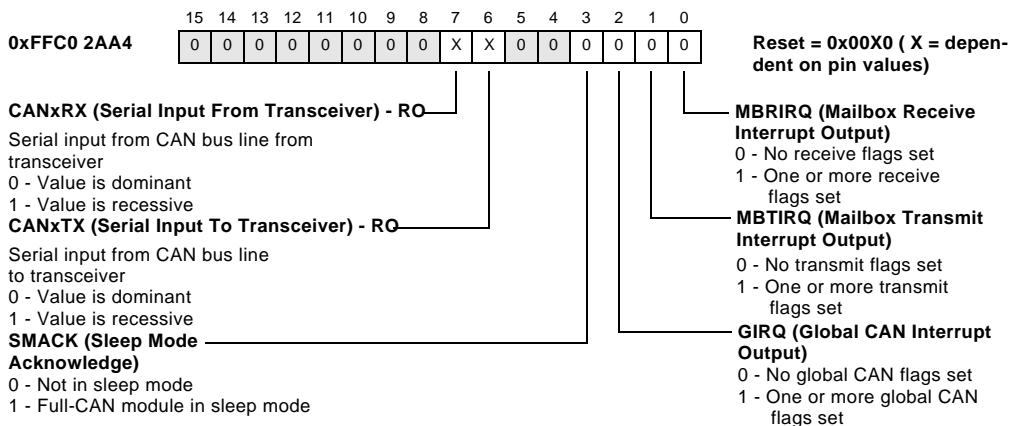


Figure 27-17. CAN Interrupt Registers

CAN Registers

CANx_GIM Global CAN Interrupt Mask Registers

Global CAN Interrupt Mask Register (CANx_GIM)

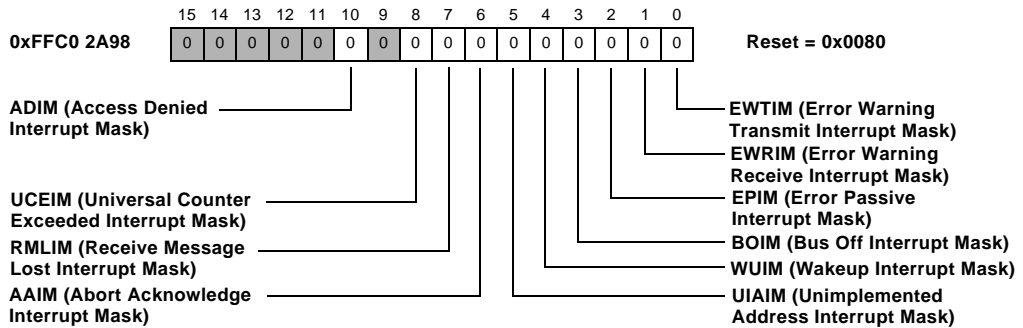


Figure 27-18. Global CAN Interrupt Mask Registers

CANx_GIS Global CAN Interrupt Status Registers

Global CAN Interrupt Status Register (CANx_GIS)

All bits are W1C

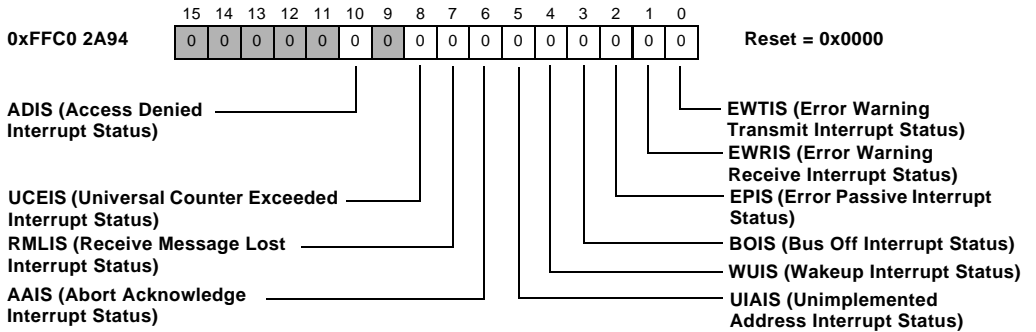


Figure 27-19. Global CAN Interrupt Status Registers

CAN Registers

CANx_GIF Global CAN Interrupt Flag Registers

Global CAN Interrupt Flag Register (CANx_GIF)

All bits W1C

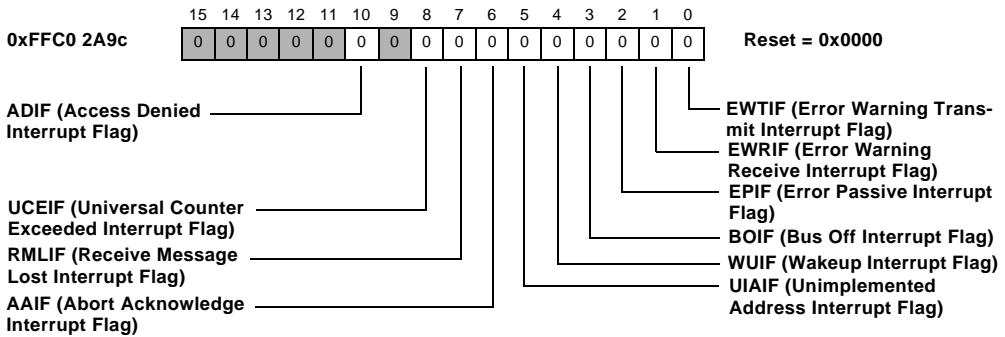


Figure 27-20. Global CAN Interrupt Flag Registers

Mailbox/Mask Registers

Figure 27-21 through Figure 27-30 on page 27-71 show the CAN mailbox and mask registers.

CANx_AMxx Acceptance Mask 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.

Acceptance Mask Register (CANx_AMxxH)

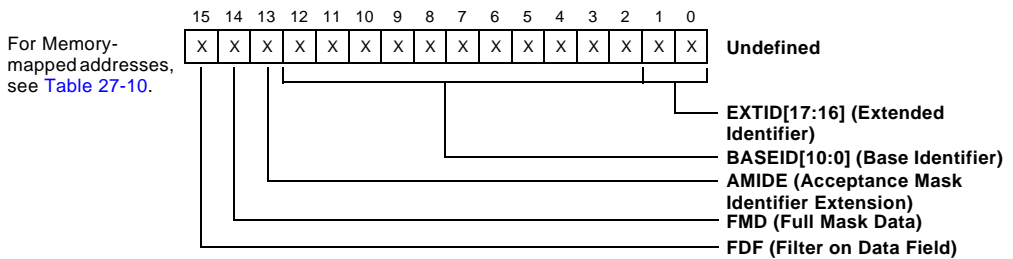


Figure 27-21. Acceptance Mask Registers (H)

Table 27-10. 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

CAN Registers

Table 27-10. Acceptance Mask Registers (H) Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address	CAN1 Memory-mapped Address
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
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

Table 27-10. Acceptance Mask Registers (H) Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address	CAN1 Memory-mapped Address
CAN _x _AM30H	0xFFC0 2BF4	0xFFC0 33F4
CAN _x _AM31H	0xFFC0 2BFC	0xFFC0 33FC

Acceptance Mask Register (CAN_x_AM_{xx}L)

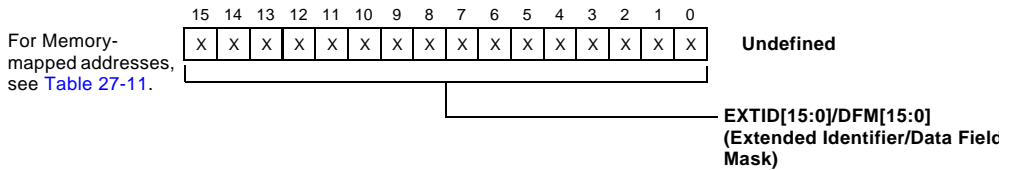


Figure 27-22. Acceptance Mask Registers (L)

Table 27-11. Acceptance Mask Registers (L) Memory-Mapped Addresses

Register Name	CAN0 Memory-mapped Address
CAN _x _AM00L	0xFFC0 2B00
CAN _x _AM01L	0xFFC0 2B08
CAN _x _AM02L	0xFFC0 2B10
CAN _x _AM03L	0xFFC0 2B18
CAN _x _AM04L	0xFFC0 2B20
CAN _x _AM05L	0xFFC0 2B28
CAN _x _AM06L	0xFFC0 2B30
CAN _x _AM07L	0xFFC0 2B38
CAN _x _AM08L	0xFFC0 2B40
CAN _x _AM09L	0xFFC0 2B48

CAN Registers

Table 27-11. Acceptance Mask Registers (L) Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address
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
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

CANx_MBxx_ID1 Registers

Mailbox Word 7 Register (CANx_MBxx_ID1)

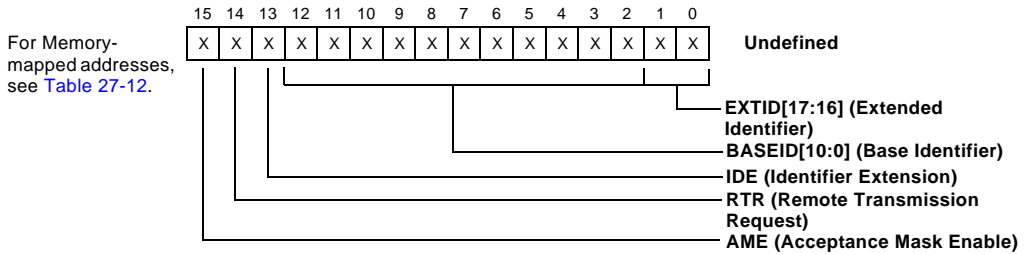


Figure 27-23. Mailbox Word 7 Register

Table 27-12. 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
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

CAN Registers

Table 27-12. Mailbox Word 7 Register Memory-Mapped Addresses (Cont'd)

Register Name	CAN0 Memory-mapped Address
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
CANx_MB29_ID1	0xFFC0 2FBC
CANx_MB30_ID1	0xFFC0 2FDC
CANx_MB31_ID1	0xFFC0 2FFC

CANx_MBxx_ID0 Registers

Mailbox Word 6 Register (CANx_MBxx_ID0)

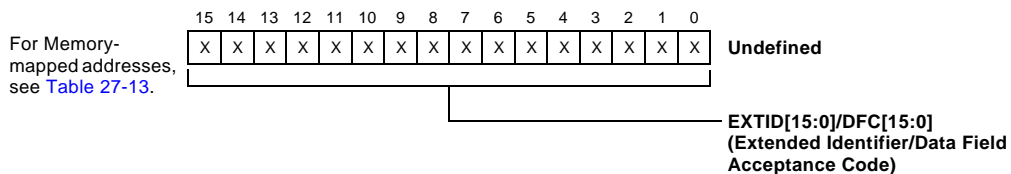


Figure 27-24. Mailbox Word 6 Register

Table 27-13. 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
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

CAN Registers

Table 27-13. Mailbox Word 6 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
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

CANx_MBxx_TIMESTAMP Registers

Mailbox Word 5 Register (CANx_MBxx_TIMESTAMP)

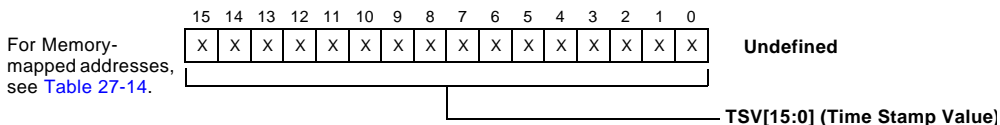


Figure 27-25. Mailbox Word 5 Register

Table 27-14. 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

CAN Registers

Table 27-14. Mailbox Word 5 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB15_TIMESTAMP	0xFFC0 2DF4
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

CANx_MBxx_LENGTH Registers

Mailbox Word 4 Register (CANx_MBxx_LENGTH)

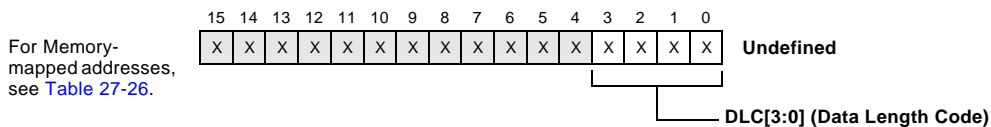


Figure 27-26. Mailbox Word 4 Register

Table 27-15. 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

CAN Registers

Table 27-15. Mailbox Word 4 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
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
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

CANx_MBxx_DATAx Registers

The following are the descriptions of Mailbox Word registers (CANx_MBxx_DATA3/2/1/0) and their appropriate memory-mapped addresses.

Mailbox Word 3 Register (CANx_MBxx_DATA3)

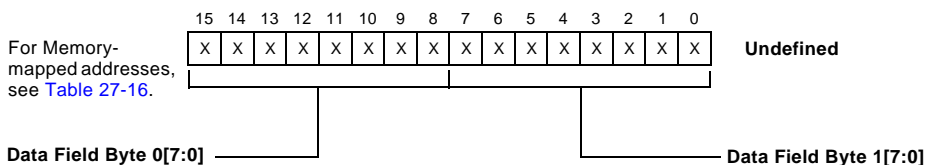


Figure 27-27. Mailbox Word 3 Register

Table 27-16. Mailbox Word 3 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CANx_MB00_DATA3	0xFFC0 2C0C
CANx_MB01_DATA3	0xFFC0 2C2C
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

CAN Registers

Table 27-16. Mailbox Word 3 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
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
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

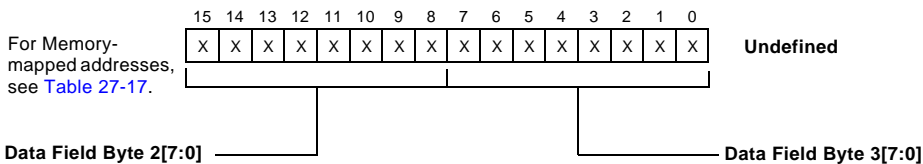
Mailbox Word 2 Register (CANx_MBxx_DATA2)

Figure 27-28. Mailbox Word 2 Register

Table 27-17. 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
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

CAN Registers

Table 27-17. Mailbox Word 2 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
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

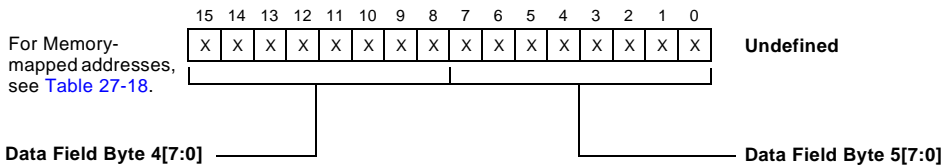
Mailbox Word 1 Register (CANx_MBxx_DATA1)

Figure 27-29. Mailbox Word 1 Register

Table 27-18. 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

CAN Registers

Table 27-18. Mailbox Word 1 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CANx_MB16_DATA1	0xFFC0 2E04
CANx_MB17_DATA1	0xFFC0 2E24
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

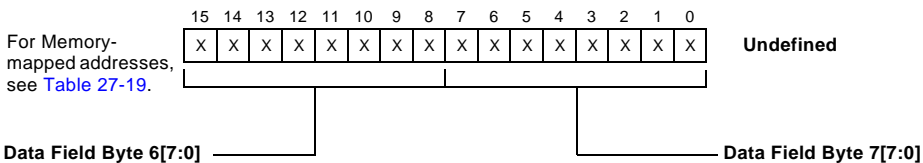
Mailbox Word 0 Register (CANx_MBxx_DATA0)

Figure 27-30. Mailbox Word 0 Register

Table 27-19. 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

CAN Registers

Table 27-19. Mailbox Word 0 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
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
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 27-31 through Figure 27-57 on page 27-86 show the mailbox control registers.

CANx_MCx Mailbox Configuration Registers

Mailbox Configuration Register 1 (CANx_MC1)

For all bits, 0 - Mailbox disabled, 1 - Mailbox enabled

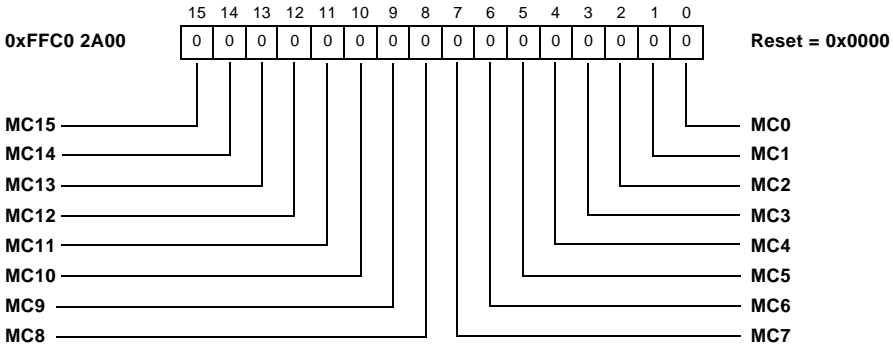


Figure 27-31. Mailbox Configuration Register 1

Mailbox Configuration Register 2 (CANx_MC2)

For all bits, 0 - Mailbox disabled, 1 - Mailbox enabled

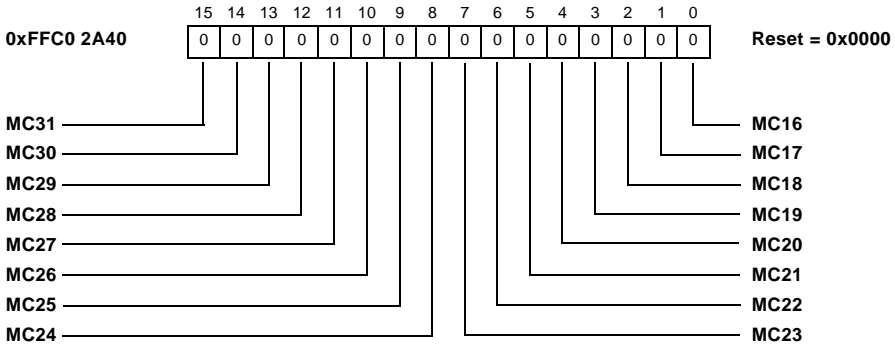


Figure 27-32. Mailbox Configuration Register 2

CAN Registers

CANx_MDx Mailbox Direction Registers

Mailbox Direction Register 1 (CANx_MD1)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

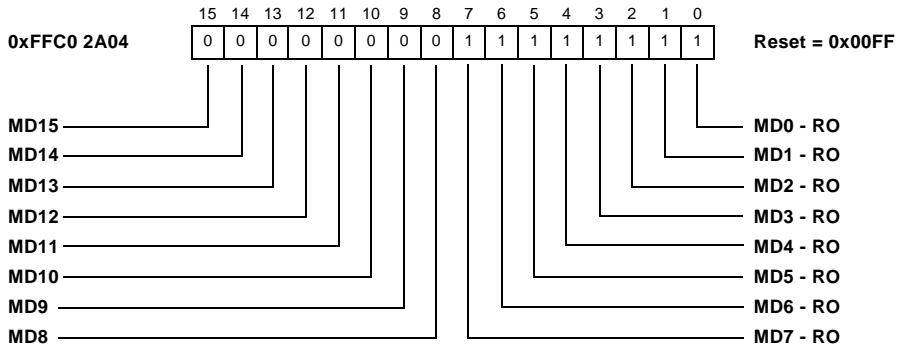


Figure 27-33. Mailbox Direction Register 1

Mailbox Direction Register 2 (CANx_MD2)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

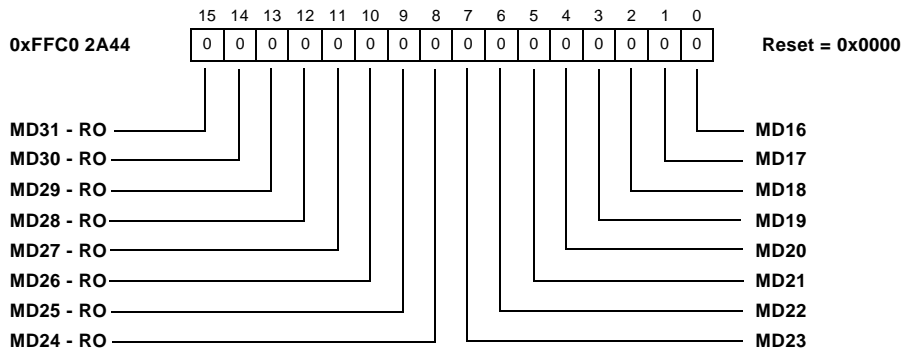


Figure 27-34. Mailbox Direction Register 2

CANx_RMPx Registers

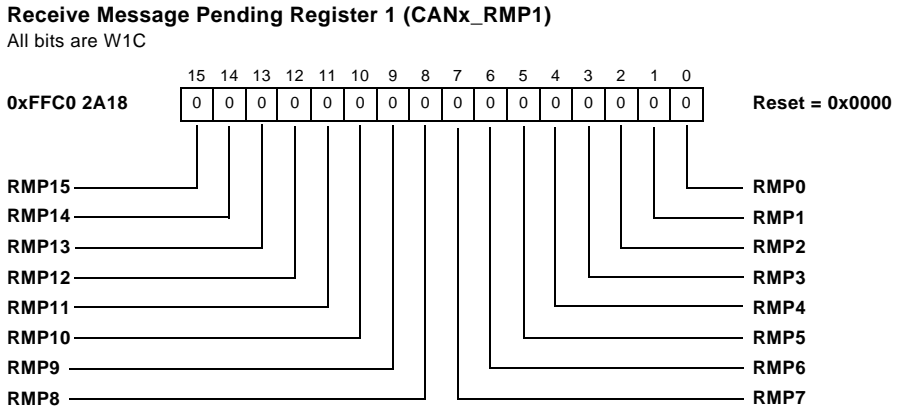


Figure 27-35. Receive Message Pending Register 1

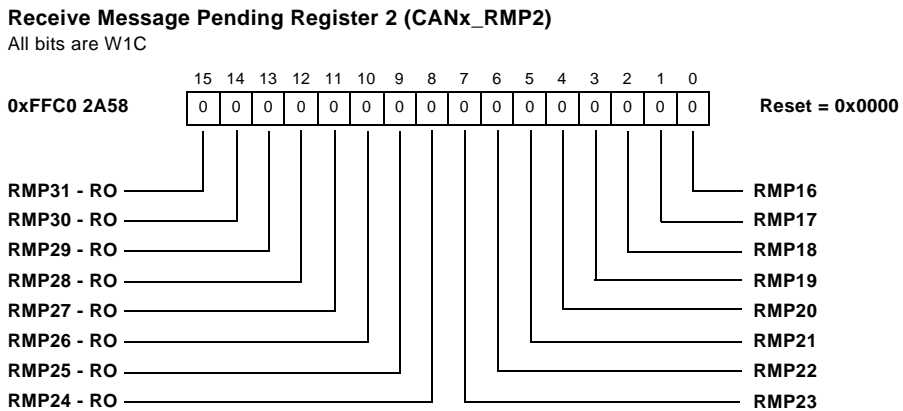


Figure 27-36. Receive Message Pending Register 2

CAN Registers

CANx_RMLx Registers

Receive Message Lost Register 1 (CANx_RML1)

RO

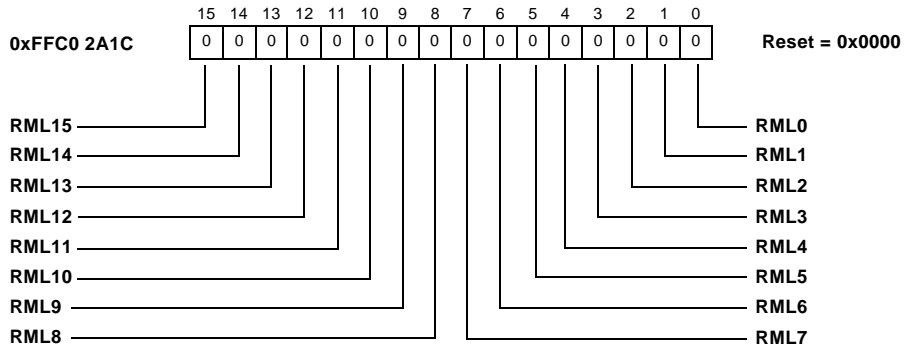


Figure 27-37. Receive Message Lost Register 1

Receive Message Lost Register 2 (CANx_RML2)

RO

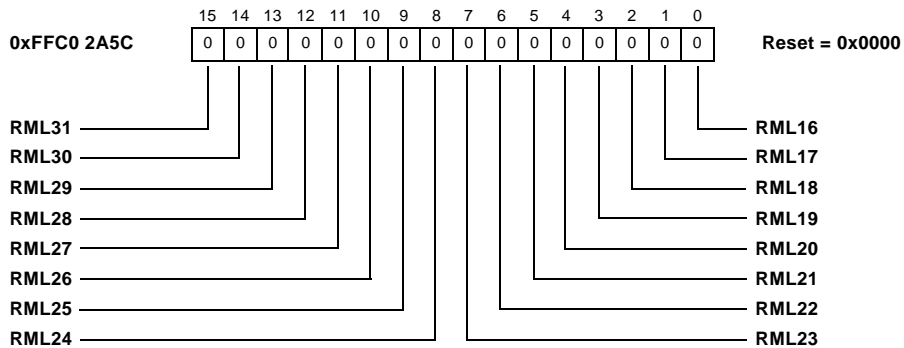


Figure 27-38. Receive Message Lost Register 2

CANx_OPSSx Register

Overwrite Protection/Single Shot Transmission Register 1

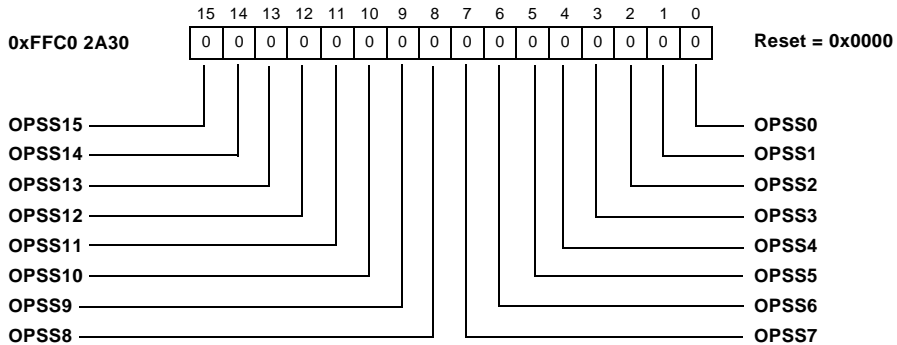


Figure 27-39. Overwrite Protection/Single Shot Transmission Register 1

Overwrite Protection/Single Shot Transmission Register 2

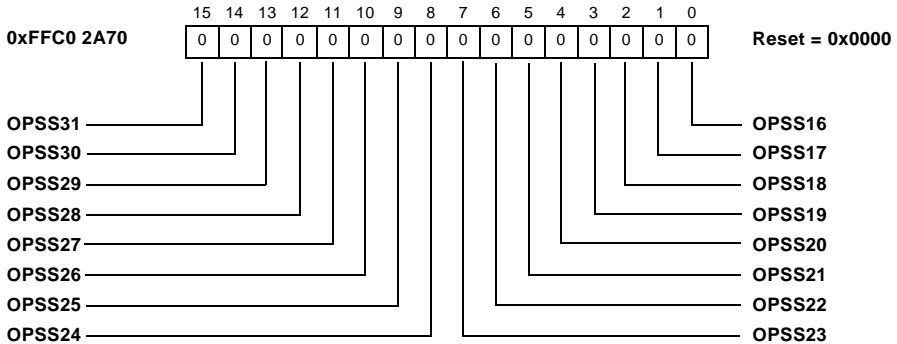


Figure 27-40. Overwrite Protection/Single Shot Transmission Register 2

CAN Registers

CANx_TRSx Registers

Transmission Request Set Register 1 (CANx_TRS1)

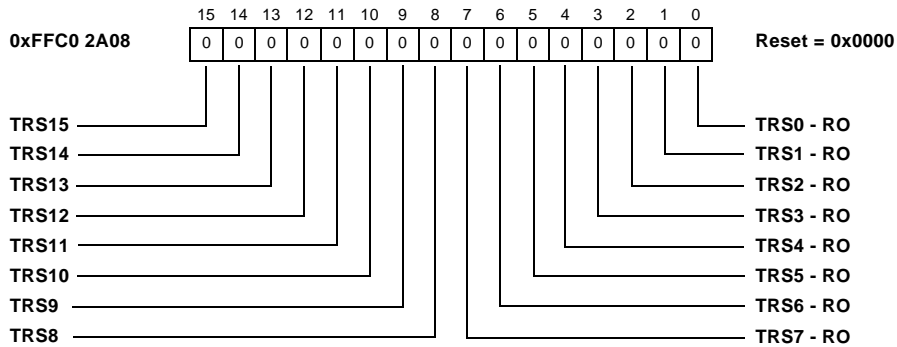


Figure 27-41. Transmission Request Set Register 1

Transmission Request Set Register 2 (CANx_TRS2)

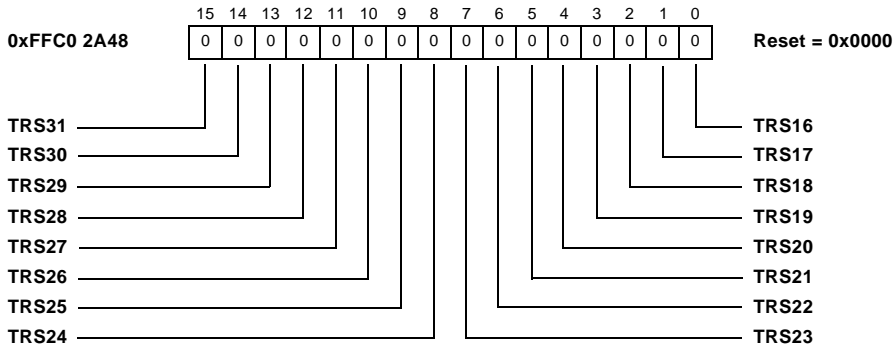


Figure 27-42. Transmission Request Set Register 2

CANx_TRRx Registers

Transmission Request Reset Register 1 (CANx_TRR1)

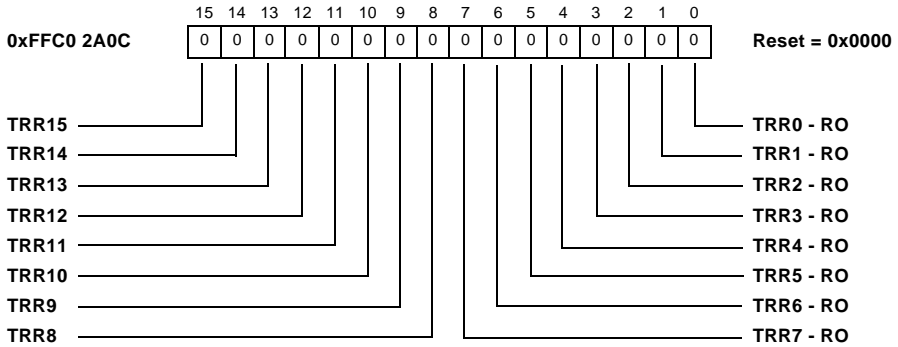


Figure 27-43. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CANx_TRR2)

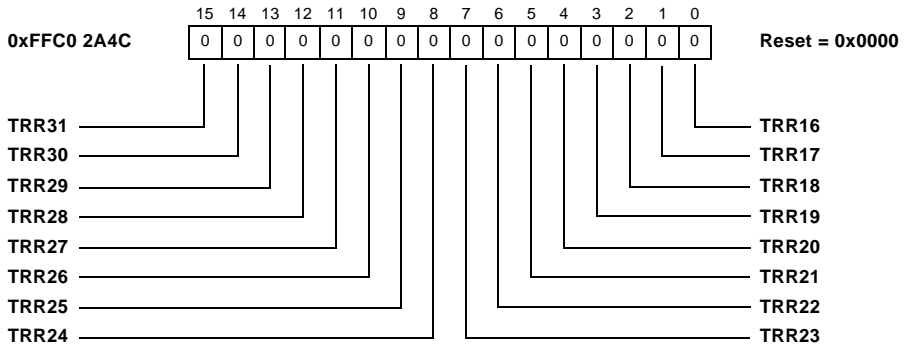


Figure 27-44. Transmission Request Reset Register 2

CAN Registers

CANx_AAx Registers

Abort Acknowledge Register 1 (CANx_AA1)

All bits are W1C

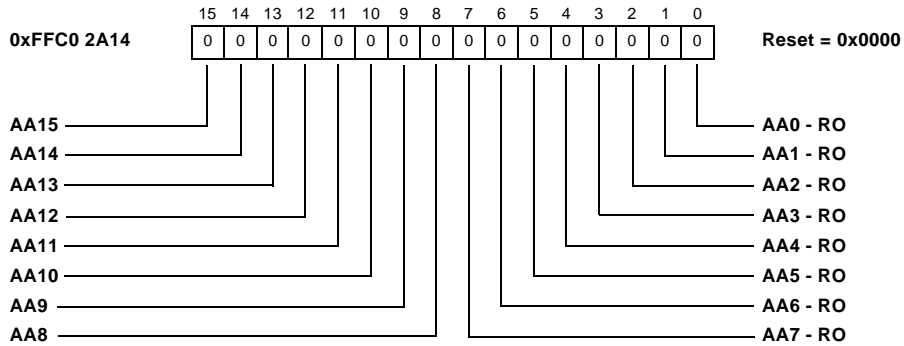


Figure 27-45. Abort Acknowledge Register 1

Abort Acknowledge Register 2 (CANx_AA2)

All bits are W1C

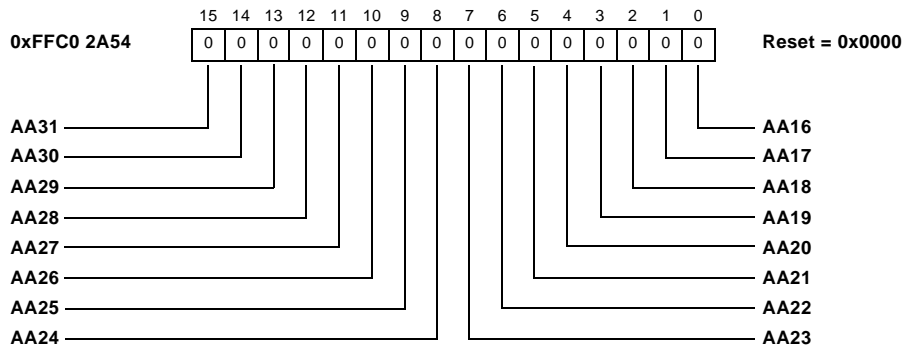


Figure 27-46. Abort Acknowledge Register 2

CANx_TAx Registers

Transmission Acknowledge Register 1 (CANx_TA1)

All bits are W1C

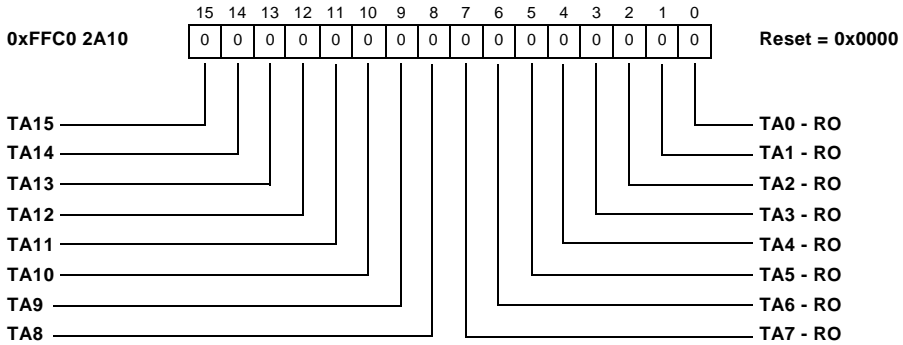


Figure 27-47. Transmission Acknowledge Register 1

Transmission Acknowledge Register 2 (CANx_TA2)

All bits are W1C

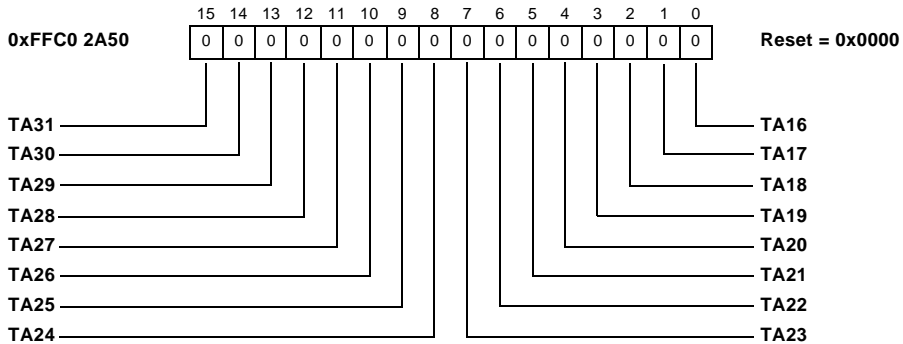


Figure 27-48. Transmission Acknowledge Register 2

CAN Registers

CANx_MBTD Register

Temporary Mailbox Disable Feature Register (CANx_MBTD)

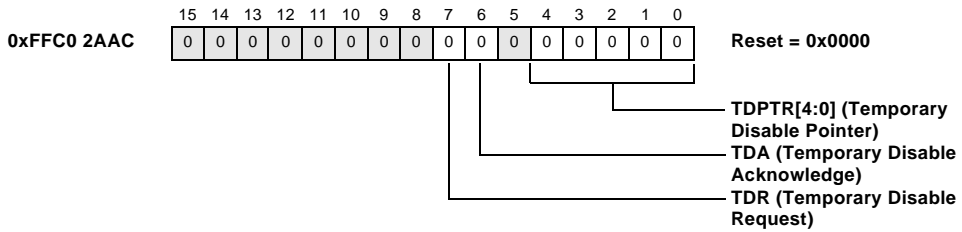


Figure 27-49. Temporary Mailbox Disable Register

CANx_RFHx Registers

Remote Frame Handling Register 1 (CANx_RFH1)

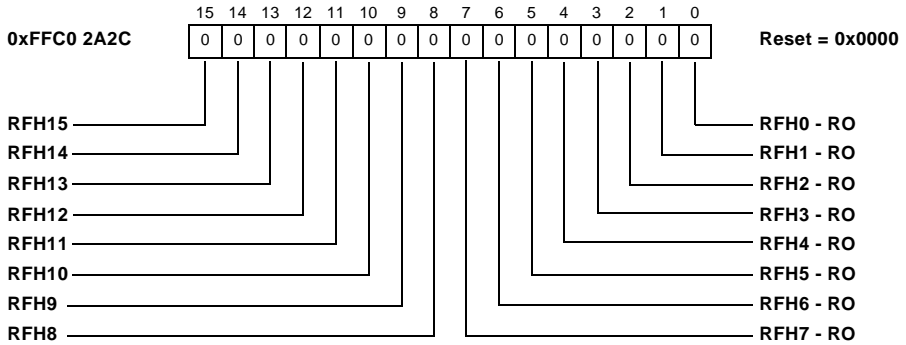


Figure 27-50. Remote Frame Handling Register 1

Remote Frame Handling Register 2 (CANx_RFH2)

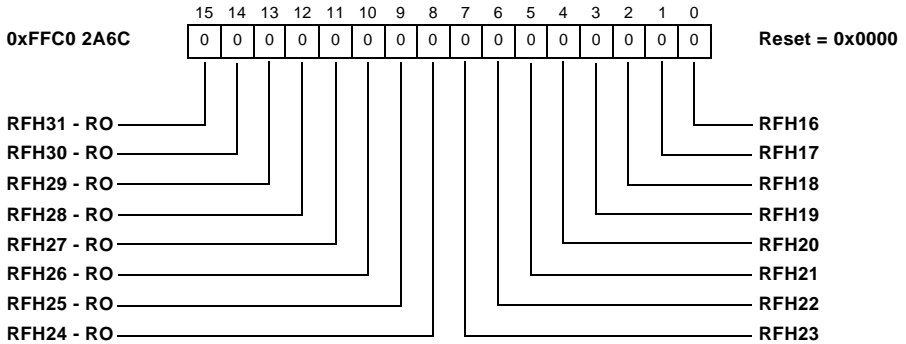


Figure 27-51. Remote Frame Handling Register 2

CAN Registers

CANx_MBIMx Registers

Mailbox Interrupt Mask Register 1 (CANx_MBIM1)

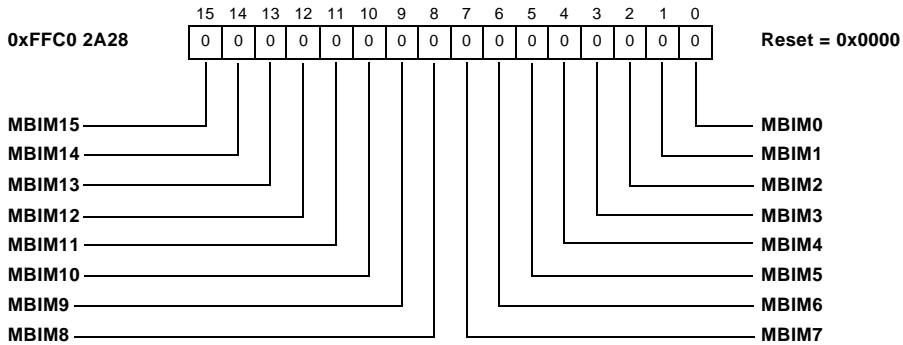


Figure 27-52. Mailbox Interrupt Mask Register 1

Mailbox Interrupt Mask Register 2 (CANx_MBIM2)

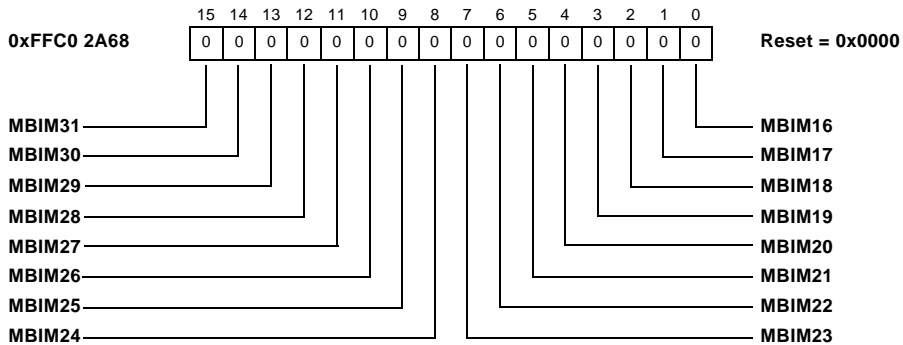


Figure 27-53. Mailbox Interrupt Mask Register 2

CANx_MBTIFx Registers

Mailbox Transmit Interrupt Flag Register 1 (CANx_MBTIF1)

All bits are W1C

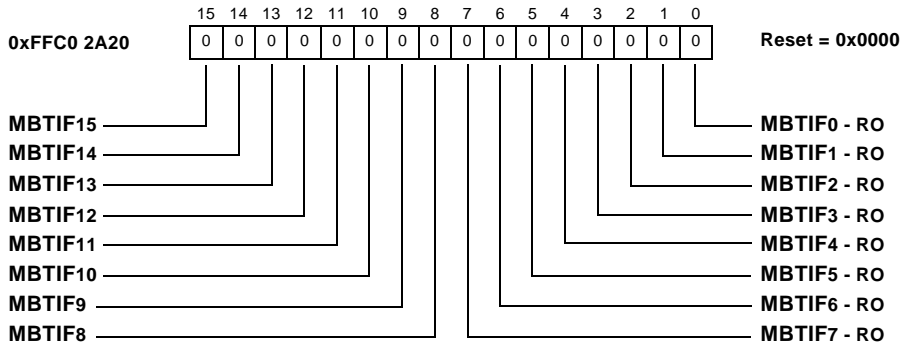


Figure 27-54. Mailbox Transmit Interrupt Flag Register 1

Mailbox Transmit Interrupt Flag Register 2 (CANx_MBTIF2)

All bits are W1C

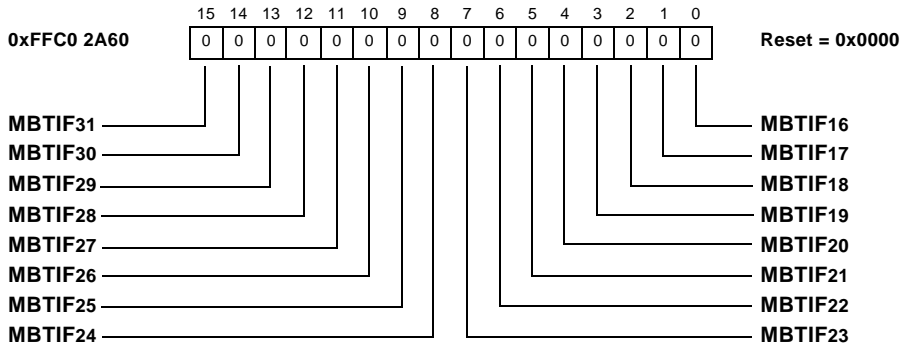


Figure 27-55. Mailbox Transmit Interrupt Flag Register 2

CAN Registers

CANx_MBRIFx Registers

Mailbox Receive Interrupt Flag Register 1 (CANx_MBRIF1)

All bits are W1C

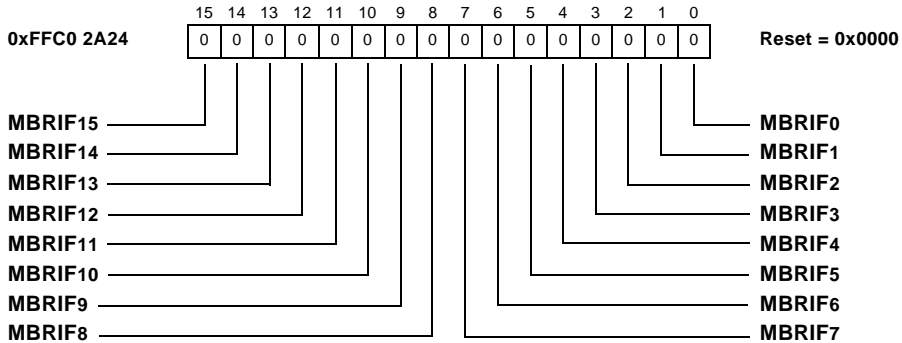


Figure 27-56. Mailbox Receive Interrupt Flag Register 1

Mailbox Receive Interrupt Flag Register 2 (CANx_MBRIF2)

All bits are W1C

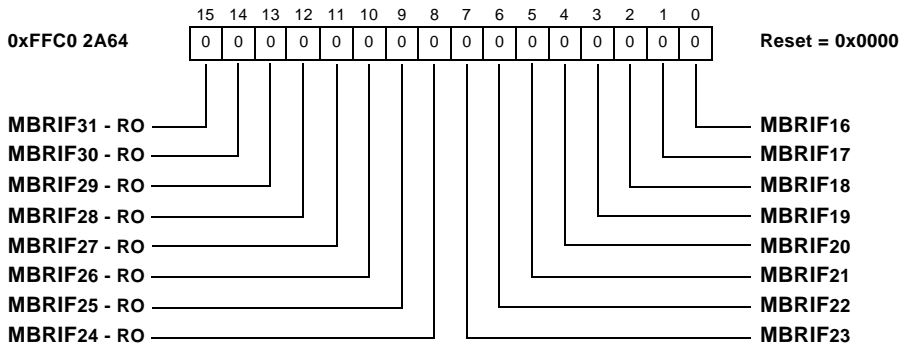


Figure 27-57. Mailbox Receive Interrupt Flag Register 2

Universal Counter Registers

Figure 27-58 through Figure 27-60 show the universal counter registers.

CANx_UCCNF Register

Universal Counter Configuration Mode Register (CANx_UCCNF)

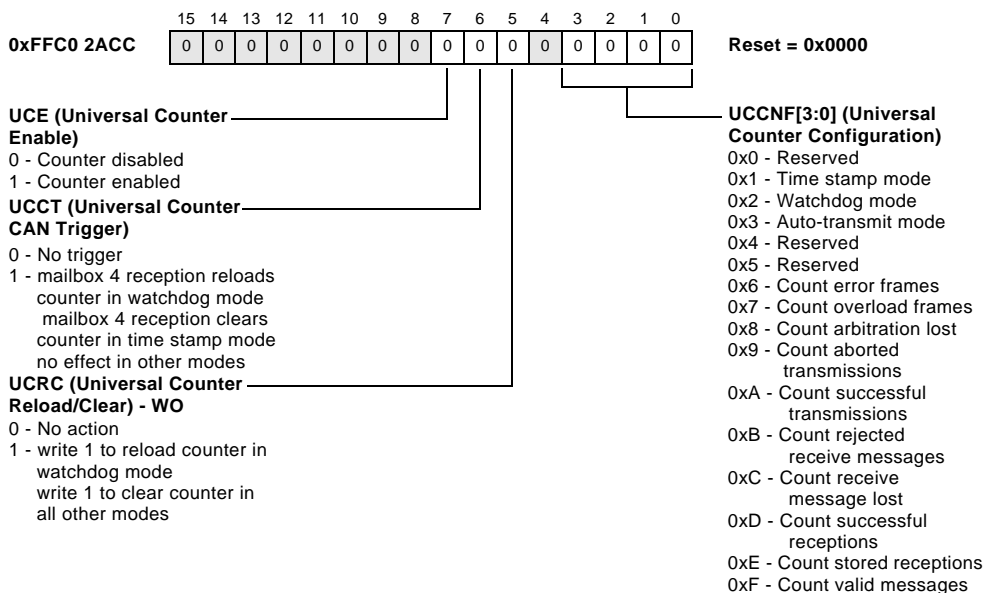


Figure 27-58. Universal Counter Configuration Mode Register

CAN Registers

CANx_UCCNT Register

Universal Counter Register (CANx_UCCNT)

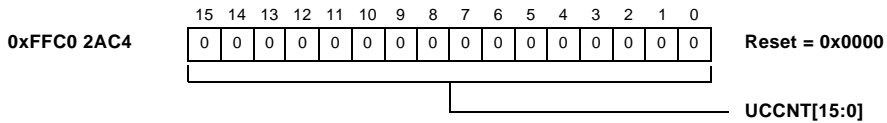


Figure 27-59. Universal Counter Register

CANx_UCRC Register

Universal Counter Reload/Capture Register (CANx_UCRC)

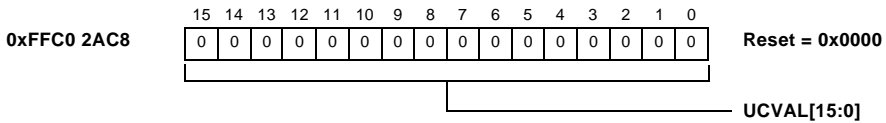


Figure 27-60. Universal Counter Reload/Capture Register

Error Registers

Figure 27-61 through Figure 27-63 show the CAN controller error registers.

CANx_CEC Register

CAN Error Counter Register (CANx_CEC)

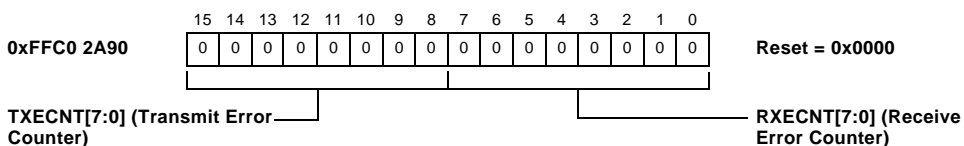


Figure 27-61. Error Counter Register

CANx_ESR Register

Error Status Register (CANx_ESR)

All bits are W1C

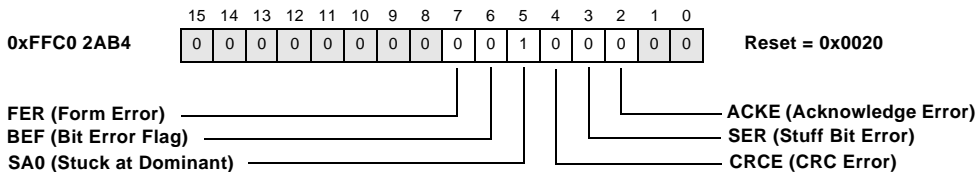


Figure 27-62. Error Status Register

CAN Registers

CANx_EWR Register

CAN Error Counter Warning Level Register (CANx_EWR)

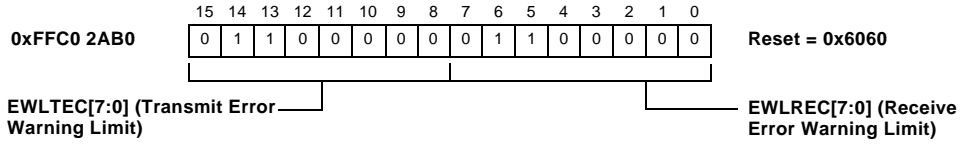


Figure 27-63. Error Counter Warning Level Register

Programming Examples

The following CAN code examples ([Listing 27-2](#) through [Listing 27-4](#) on [page 27-95](#)) 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 27-2. Initializing CAN0

```
Initialize_CAN0:
    PO.H = HI(PORTG_FER); /* CAN pins multiplexed on Port G */
    PO.L = LO(PORTG_FER);
    RO = 0x3000 (Z); /* Enable CAN0 TX/RX pins on PG12/PG13 */
    W[PO] = RO;
    SSYNC;

    /* =====
    ** Set CAN Bit Timing
    **
    ** CANx_TIMING - SJW, TSEG2, and TSEG1 governed by:
    ** SJW <= TSEG2 <= TSEG1
    **
    ** =====
    */
    PO.H = HI(CAN0_TIMING);
    PO.L = LO(CAN0_TIMING);
```

Programming Examples

```
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 27-3. Initializing and Enabling Mailboxes

```

CAN0_Initialize-Mailboxes:
    PO.H = HI(CAN0_MD1); /* Configure Mailbox Direction */
    PO.L = LO(CAN0_MD1);
    RO = W[PO](Z);
    BITCLR(RO, BITPOS(MD8)); /* Set MB08 for Transmit */
    BITSET(RO, BITPOS(MD9)); /* Set MB09 for Receive */
    W[PO] = RO;
    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 */
    RO = 0x411 << 2; /* Put Message ID in correct slot */
    PO.L = LO(CAN0_MB_ID1(8)); /* Access MB08 ID1 Register */
    W[PO] = RO; /* Remote frame disabled, 11 bit ID */

    RO = 0;
    PO.L = LO(CAN0_MB_ID0(8));
    W[PO] = RO; /* Zero Out Lower ID Register */

    RO = 4;
    PO.L = LO(CAN0_MB_LENGTH(8));

```

Programming Examples

```
W[PO] = R0; /* Set DLC to 4 Bytes */

R0 = 0xAABB(Z);
PO.L = LO(CANO_MB_DATA3(8));
W[PO] = R0; /* Byte0 = 0xAA, Byte1 = 0xBB */

R0 = 1;
PO.L = LO(CANO_MB_DATA2(8));
W[PO] = R0; /* Initialize Count to 1 */

/* Initialize Mailbox 9 For Receive */
R0 = 0x007 << 2; /* Put Message ID in correct slot */
PO.L = LO(CANO_MB_ID1(9)); /* Access MB08 ID1 Register */
W[PO] = R0; /* Remote frame disabled, 11 bit ID */

R0 = 0;
PO.L = LO(CANO_MB_ID0(9));
W[PO] = R0; /* Zero Out Lower ID Register */
SSYNC;

/* Enable the Configured Mailboxes */
PO.L = LO(CANO_MC1);
R0 = W[PO](Z);
BITSET(R0, BITPOS(MC8)); /* Enable MB08 */
BITSET(R0, BITPOS(MC9)); /* Enable MB09 */
W[PO] = 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 `CANO_RX_HANDLER` and `CANO_TX_HANDLER` have been properly registered in the system interrupt controller and that the interrupts are enabled properly in the `SIC_IMASK0` register.

Listing 27-4. CAN Transfers and Interrupts

```

CAN0_SetupIRQs_and_Transfer:
    PO.H = HI(CANO_MBIM1);
    PO.L = LO(CANO_MBIM1);
    RO = 0;
    BITSET(RO, BITPOS(MBIM8)); /* Enable Mailbox Interrupts */
    BITSET(RO, BITPOS(MBIM9)); /* for Mailboxes 8 and 9 */
    W[PO] = RO;
    SSYNC;
    /* Leave CAN Configuration Mode (Clear CCR) */
    PO.L = LO(CANO_CONTROL);
    RO = W[PO](Z);
    BITCLR(RO, BITPOS(CCR));
    W[PO] = RO;

    PO.L = LO(CANO_STATUS);
/* Wait for CAN Configuration Acknowledge (CCA) */
    WAIT_FOR_CCA_TO_CLEAR:
        R1 = W[PO](Z);
        CC = BITTST (R1, BITPOS(CCA));
        IF CC JUMP WAIT_FOR_CCA_TO_CLEAR;
    PO.L = LO(CANO_TRS1);
    RO = TRS8; /* Transmit Request MB08 */
    W[PO] = RO; /* Issue Transmit Request */
    SSYNC;
    Wait_Here_For_IRQs:
        NOP;
        NOP;
        NOP;

```

Programming Examples

```
        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;

```

Programming Examples

28 SPI-COMPATIBLE PORT CONTROLLERS

This chapter describes the serial peripheral interface (SPI) ports and includes the following sections:

- [“Overview” on page 28-1](#)
- [“Interface Overview” on page 28-3](#)
- [“Description of Operation” on page 28-16](#)
- [“Functional Description” on page 28-26](#)
- [“Programming Model” on page 28-30](#)
- [“SPI Registers” on page 28-43](#)
- [“Programming Examples” on page 28-50](#)

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.

Overview

SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. [Table 28-1](#) lists the critical SPI signals.

Table 28-1. 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 28-1 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.

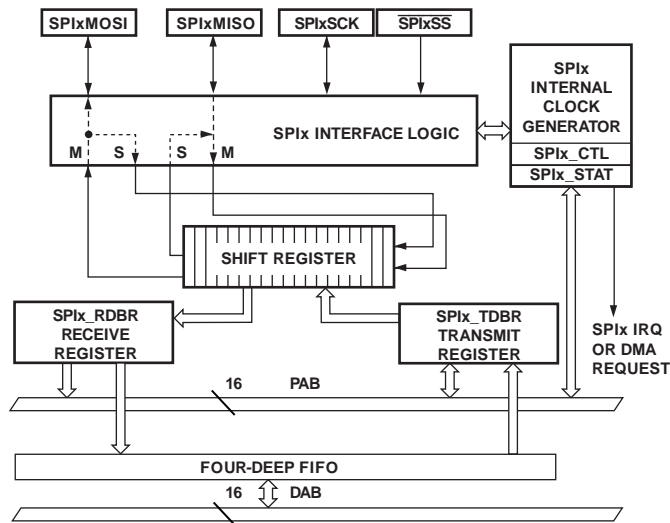


Figure 28-1. SPIx Block Diagram

External Interface

All of the SPI signals are accessible through GPIO ports. The four SPI signals that make up the 4-wire interface (SPI_xSCK, SPI_xMISO, SPI_xMOSI, and $\overline{\text{SPI}}_{\text{x}}\text{SS}$) 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 PORT_x_FER register.

If the configurable pin is shared among multiple peripherals, the associated PORT_x_MUX register will also need to be written to explicitly configure it as SPI. Table 28-2 provides a mapping of SPI pins to GPIO pins along with an explanation regarding how to enable these pins for use as SPI.

Table 28-2. 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
$\overline{\text{SPI}}_{0}\text{SS}$	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
$\overline{\text{SPI}}_{1}\text{SS}$	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
$\overline{\text{SPI}}_{2}\text{SS}$	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” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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 `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 28-17. 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” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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” in the *ADSP-BF54x Blackfin Processor Hardware Reference*.

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” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

i Only one slave is allowed to transmit data at any given time.

The SPI configuration example in [Figure 28-2](#) illustrates how the processor can be used as the slave SPI device. The 8-bit host micro controller is the SPI master.

i The processor can be booted by way of its SPI interface to allow user application code and data to be downloaded before runtime.

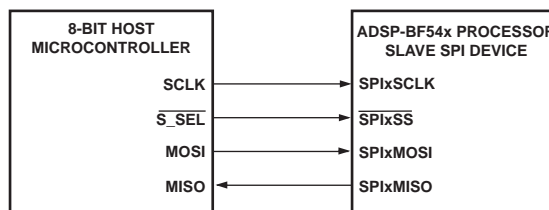


Figure 28-2. ADSP-BF54x as Slave SPI Device

Serial Peripheral Interface Slave Select Input Signal

The $\overline{\text{SPIxSS}}$ 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 $\overline{\text{SPIxSS}}$ 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.

Interface Overview

The $\overline{\text{SPIxSS}}$ 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” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

The enable lead time (T_1), the enable lag time (T_2), and the sequential transfer delay time (T_3) each must always be greater than or equal to one-half the SPIxSCK period. See [Figure 28-3](#). The minimum time between successive word transfers (T_4) 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).

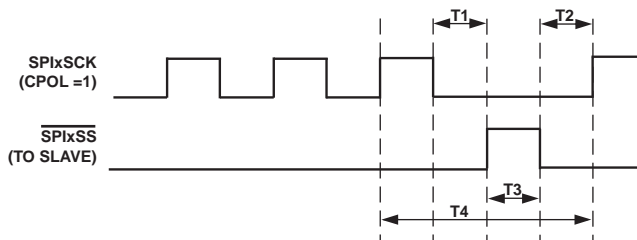


Figure 28-3. SPI Timing

For a master device with $\text{CPHA} = 0$, the slave select output is inactive (high) for at least one-half the SPIxSCK period. In this case, T_1 and T_2 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 28-46](#) 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 28-3](#) summarizes how to setup the port control logic in order to enable the individual slave select enable outputs.

Table 28-3. SPI Slave Select Enable Setup

Signal Name	Pin Name	Port Control To Enable Signal
$\overline{\text{SPIOSEL1}}$	PE4	Set bit 4 in PORTE_FER = 1 Set PORTE_MUX[9:8] = b#00
$\overline{\text{SPIOSEL2}}$	PE5	Set bit 5 in PORTE_FER = 1 Set PORTE_MUX[11:10] = b#00
$\overline{\text{SPIOSEL3}}$	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 4 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

Interface Overview

Table 28-3. SPI Slave Select Enable Setup (Cont'd)

Signal Name	Pin Name	Port Control To Enable Signal
$\overline{\text{SPI2SEL2}}$	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 SPI_x_FLG register to enable up to three general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the PB_x, PE_x, and PG_x 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 PB_x, PE_x, and PG_x pins must be enabled for use by SPI in the appropriate PORT_x_FER and PORT_x_MUX registers. For more information, see Chapter 9, “General-Purpose Ports” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Refer to [Table 28-4](#) for more details regarding which port pins must be configured prior to being modified by way of the SPI_x_FLG register.

In slave mode, the SPI_x_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 PORT_x_MUX register, and then as an SPI pin in the PORT_x_FER register. [Figure 28-14 on page 28-46](#) shows the SPI_x_FLG register diagram.

Table 28-4. SPI_x_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

Table 28-4. SPIx_FLG Bit Mapping to Port Pins (Cont'd)

Bit	Name	Function	Port Pin	Default
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

Interface Overview

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. The processor can:

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.

In case 3, all four devices connected by way of SPI ports can be other processors.

3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the two other slave processors) at a time and transmit broadcast data to all three at the same time. This EMISO feature may be avail-

able in some other microcontrollers. Therefore, it is possible to use the $\overline{\text{EMISO}}$ feature with any other SPI device that includes this functionality.

Figure 28-4 shows one processor as a master with three other SPI-compatible devices as slaves.

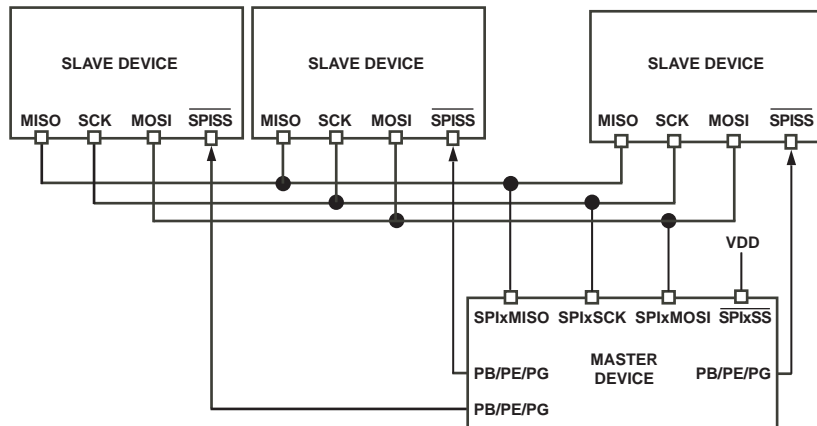


Figure 28-4. Single-Master, Multiple-Slave Configuration

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.

i 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.

Interface Overview

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 and the SPI can be disabled or enabled for another mode.

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” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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.

i 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 and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

SPI Transmit Data Buffer

The `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.

Description of Operation

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 28-5](#).

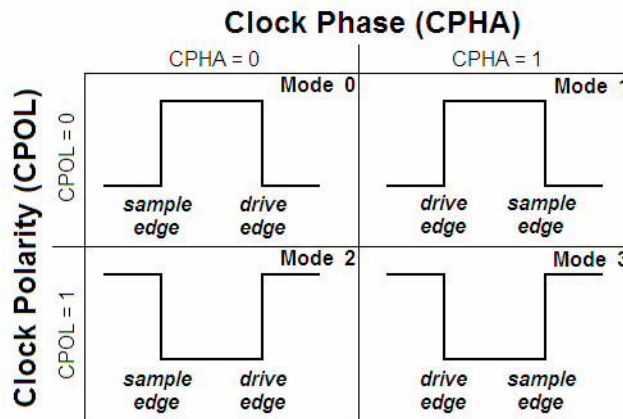


Figure 28-5. SPI Modes of Operation

The figures “[SPI Transfer Protocol for CPHA = 0](#)” on page 28-18 and “[SPI Transfer Protocol for CPHA = 1](#)” on page 28-19 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 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

Description of Operation

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 28-6 shows the SPI transfer protocol for `CPHA` = 0. Note `SPIxSCK` starts toggling in the middle of the data transfer, `SIZE` = 0, and `LSBF` = 0.

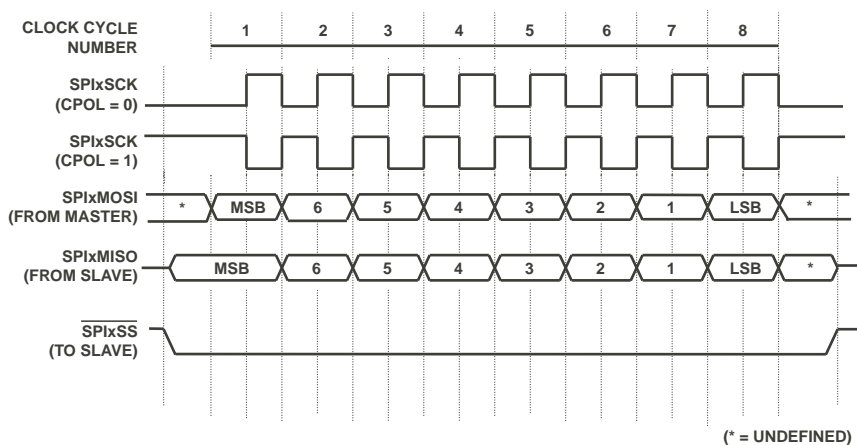


Figure 28-6. SPI Transfer Protocol for `CPHA` = 0

Figure 28-7 shows the SPI transfer protocol for $CPHA = 1$. Note $SPIxSCK$ starts toggling at the beginning of the data transfer, $SIZE = 0$, and $LSBF = 0$.

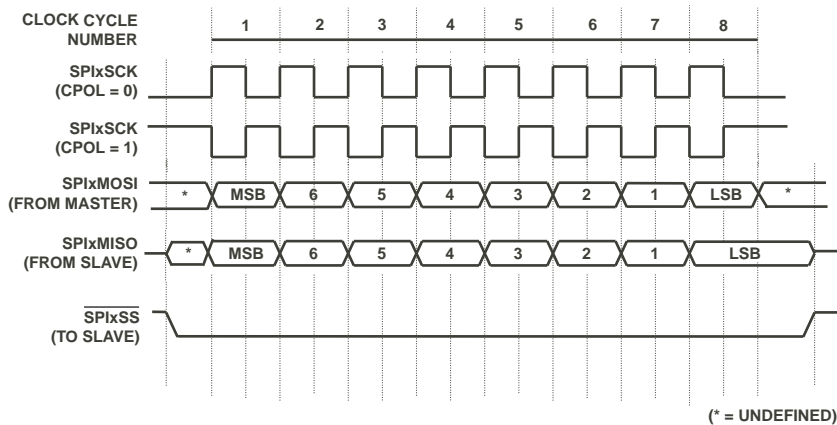


Figure 28-7. SPI Transfer Protocol for $CPHA = 1$

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 con-

Description of Operation

sists of a single master and a single slave, $C_{PHA} = 1$, 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 (SPI_{xMOSI} and SPI_{xMISO}) 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 SPI_{xMOSI} and SPI_{xMISO} 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 SPI_{xMOSI} pin is three-stated when the data driven out on SPI_{xMOSI} is a logic high. The SPI_{xMOSI} 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 SPI_{xMISO} pin is three-stated if the data driven out on SPI_{xMISO} 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 (SPI_{xSS}). 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 SPI_{xMOSI} pins are connected together, all SPI_{xMISO} pins are connected together, and all SPI_{xSCK} 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 28-4 on page 28-10](#).

 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.

Description of Operation

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 28-24](#).

[Figure 28-13 on page 28-45](#) 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 28-5 on page 28-17](#).

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:

$$\text{SPIxSCK Frequency} = \frac{\text{SCLK System Clock Frequency}}{2 \times \text{SPIx_BAUD}}$$

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 28-5 lists several possible baud rate values for `SPIx_BAUD`.

Table 28-5. 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.

Description of Operation

See [Figure 28-15 on page 28-48](#) 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, SPIx_MISO, and SPIx_SCK) are disabled. However, the slave select output pins revert to being controlled by the general-purpose I/O port registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the general-purpose I/O port registers appropriately.

When enabling the `MODF` feature, the program must configure as inputs all of the port pins that will be used as slave selects. Programs can do this by configuring the direction of the port pins prior to configuring the SPI. This ensures that, once the `MODF` error occurs and the slave selects are automatically reconfigured as port pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The `TXE` bit is set in `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.

Functional Description

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 28-21](#).

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 28-8 on page 28-39](#) 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 28-6](#).


 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 28-6. 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 28-8 on page 28-39](#) 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 $\text{SZ} = 1$ and the transmit buffer is empty, the device repeatedly transmits 0s on the SPIxMISO pin. If $\text{SZ} = 0$ and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If $\text{GM} = 1$ and the receive buffer is full, the device contin-

Programming Model

ues 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 28-7](#) are necessary to prepare the device for a new transfer.

Table 28-7. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and Receive	Interrupt active when receive buffer is full. Read of <code>SPIx_RDBR</code> clears interrupt.
b#01	Transmit and Receive	Interrupt active when transmit buffer is empty. Writing to <code>SPIx_TDBR</code> clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as <code>SPIx</code> DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as <code>SPIx</code> 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{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{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

Programming Model

`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 5, “Direct Memory Access” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.
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

Programming Model

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 28-9 on page 28-40](#) 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 `SPIxSS` signal to the active-low state or by the first active edge of `SPIxSCK`, depending on the state of `CPHA`.

Programming Model

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 5, “Direct Memory Access” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.
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 28-9 on page 28-40](#) 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

Programming Model

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`).

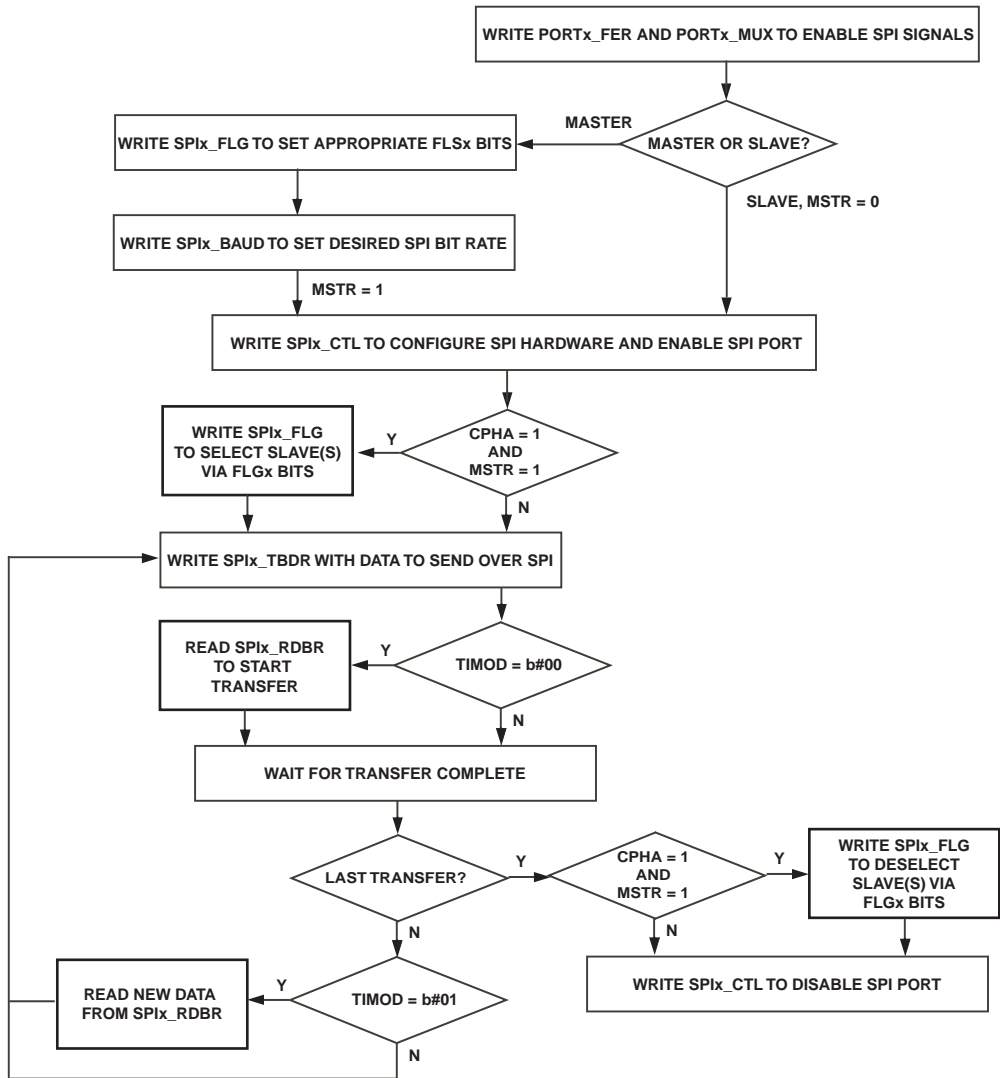


Figure 28-8. Core-Driven SPI Flow Chart

Programming Model

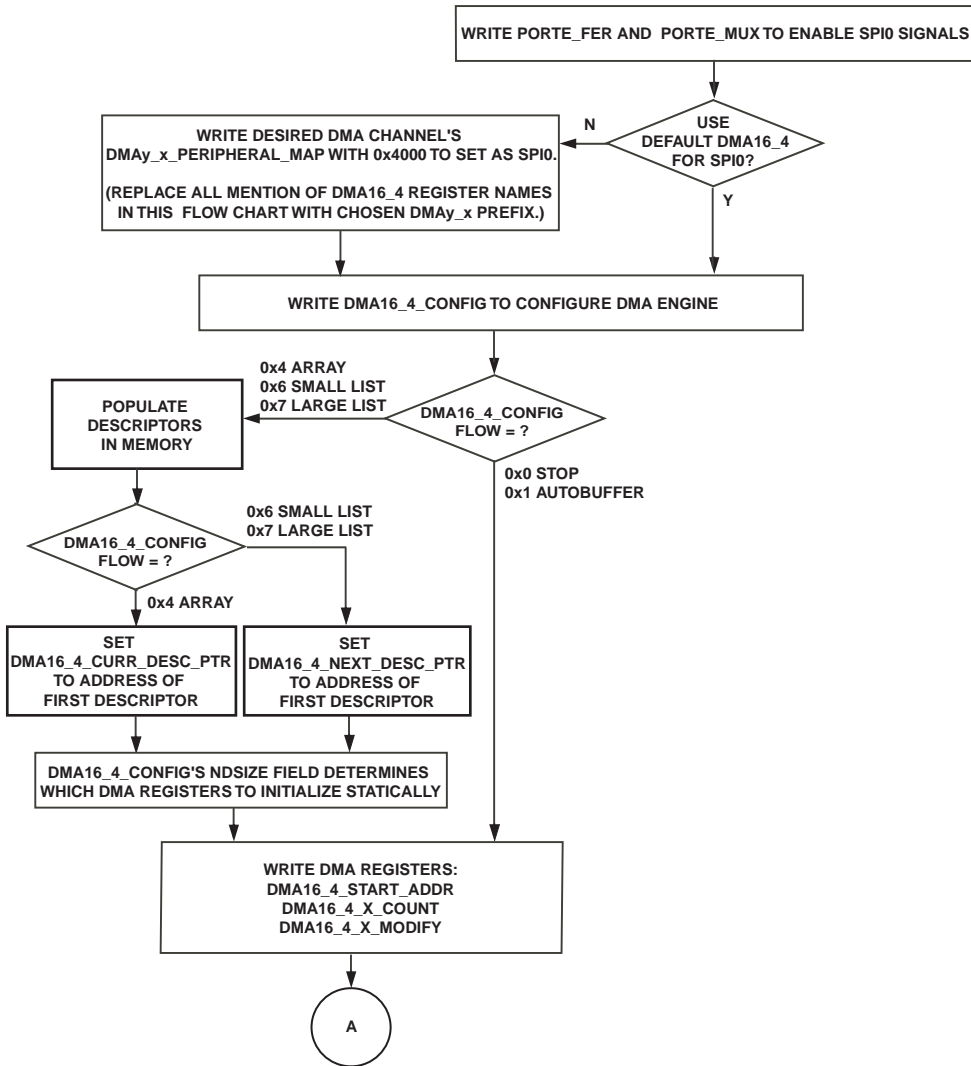


Figure 28-9. SPI0 DMA Flow Chart (Part 1 of 3)

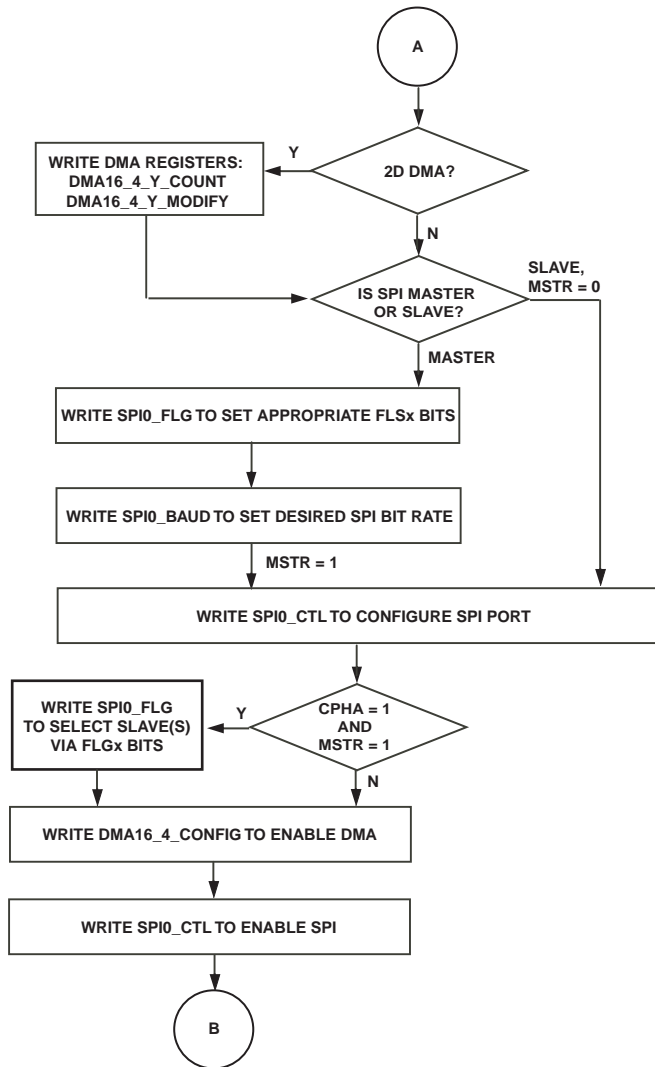


Figure 28-10. SPI0 DMA Flow Chart (Part 2 of 3)

Programming Model

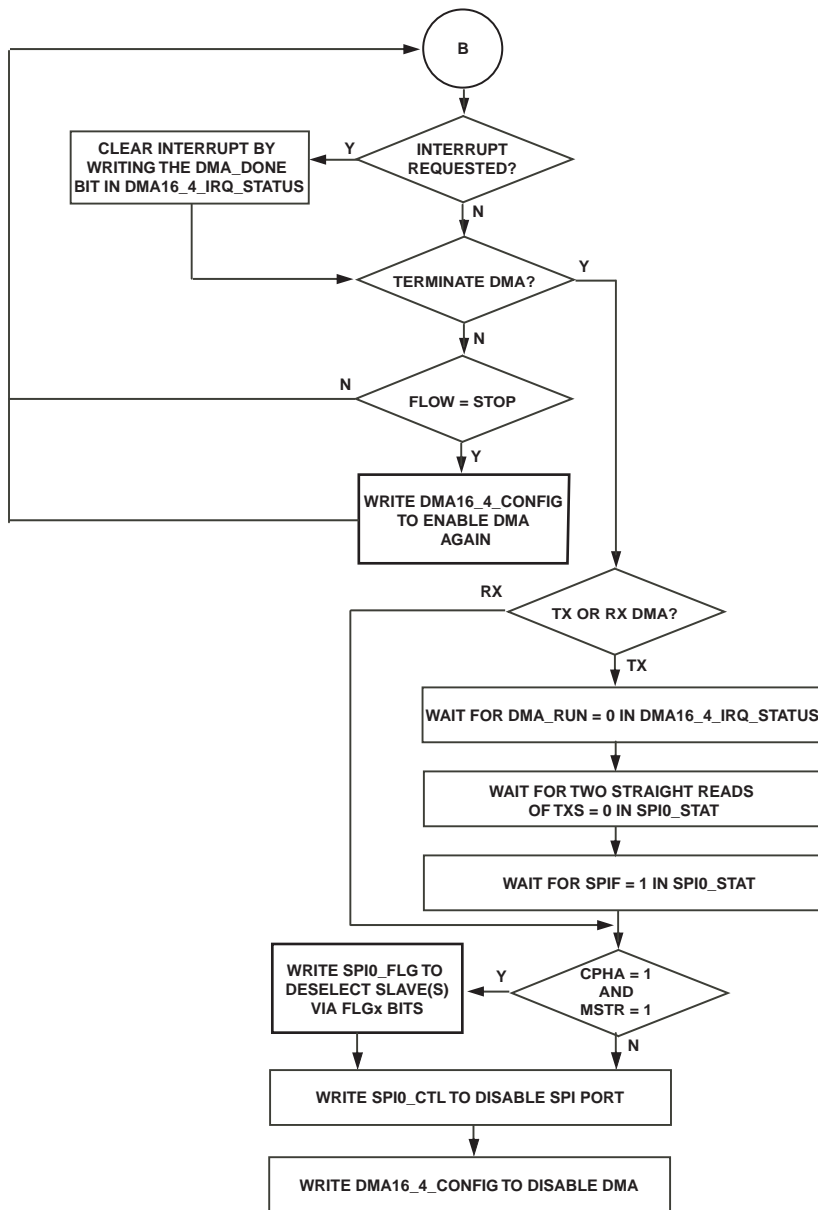


Figure 28-11. SPI0 DMA Flow Chart (Part 3 of 3)

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 information about DMA-related registers, see Chapter 5, “Direct Memory Access” in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*. The internal shift register, SFDRx, is not directly accessible.

See “Error Signals and Flags” on page 28-23 for more information about how the bits in these registers are used to signal errors and other conditions.

Table 28-8 shows the functions of the SPI registers. Figure 28-12 through Figure 28-18 on page 28-49 provide details.

Table 28-8. SPI Registers

Register Name	Function	Notes
SPIx_BAUD	SPIx port baud rate registers on page 28-44	Value of 0 or 1 disables the serial clock
SPIx_CTL	SPIx port control registers on page 28-45	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPIx_FLG	SPIx port flag registers on page 28-46	Bits 0 and 8 are reserved
SPIx_STAT	SPIx port status registers on page 28-48	SPIF bit can be set by clearing SPE in SPIx_CTL
SPIx_TDBR	SPIx port transmit data buffer registers on page 28-48	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPIx_CTL)

SPI Registers

Table 28-8. SPI Registers (Cont'd)

Register Name	Function	Notes
SPIx_RDBR	SPIx port receive data buffer registers on page 28-49	When register is read, hardware events can be triggered
SPIx_SHADOW	SPIx port RDBR shadow registers on page 28-49	Register has the same contents as SPIx_RDBR, but no action is taken when it is read

SPI Baud Rate (SPIx_BAUD) Register

SPI Baud Rate Register (SPIx_BAUD)

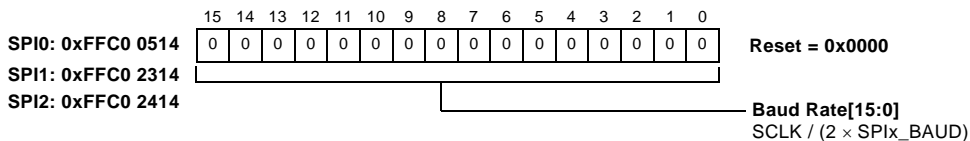


Figure 28-12. SPI Baud Rate Register

SPI Control (SPIx_CTL) Register

SPI Control Register (SPIx_CTL)

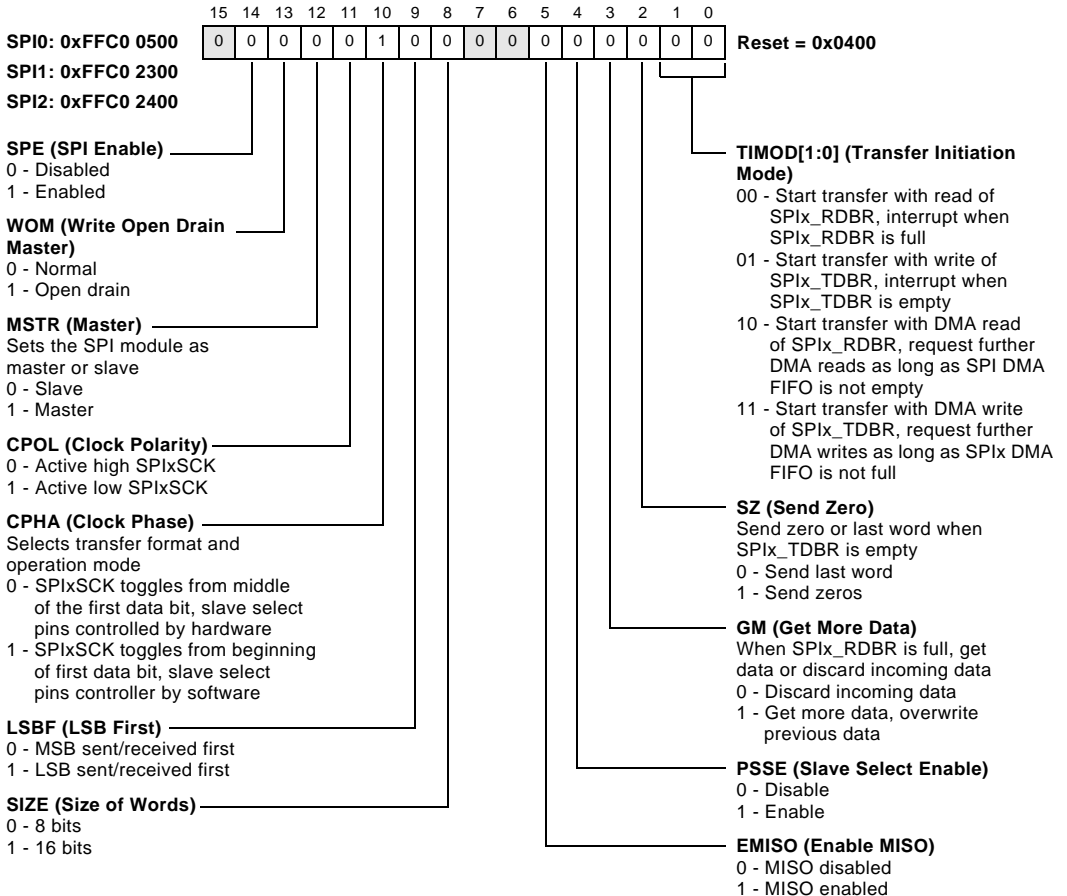


Figure 28-13. SPI Control Register

SPI Flag (SPIx_FLG) Register

SPI Flag Register (SPIx_FLG)

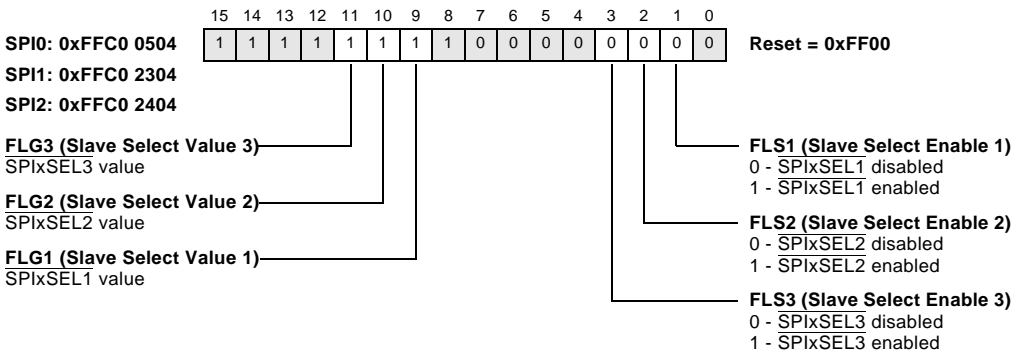


Figure 28-14. SPIx Flag Register

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 (SPI0SEL1). [Table 28-4 on page 28-10](#) 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 `SPI0_FLG` must be set. Clearing `FLG3` in `SPI0_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 `SPI0_FLG`. It is not necessary to write to the `FLG3` bit, because the SPI hardware automatically drives the `PE6` pin.

SPI Registers

SPI Status (SPIx_STAT) Register

SPI Status Register (SPIx_STAT)

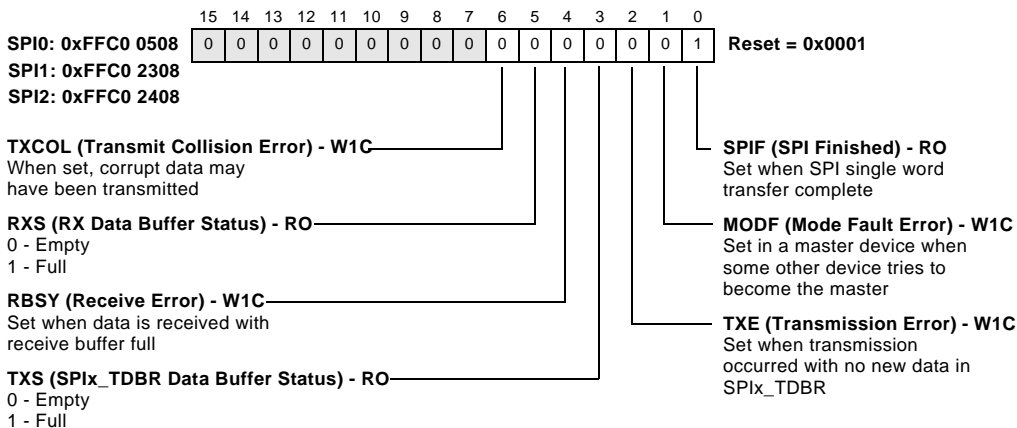


Figure 28-15. SPI Status Register

SPI Transmit Data Buffer (SPIx_TDBR) Register

SPI Transmit Data Buffer Register (SPIx_TDBR)

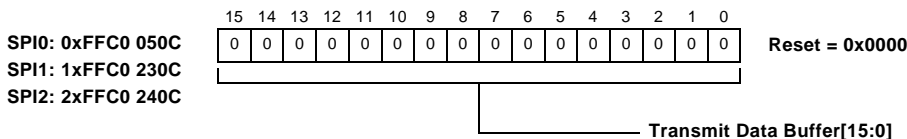


Figure 28-16. SPI Transmit Data Buffer Register

SPI Receive Data Buffer (SPIx_RDBR) Register

SPI Receive Data Buffer Register (SPIx_RDBR)

RO

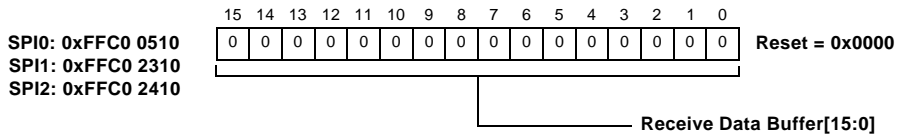


Figure 28-17. SPI Receive Data Buffer Register

SPI RDBR Shadow (SPIx_SHADOW) Register

SPI RDBR Shadow Register (SPIx_SHADOW)

RO

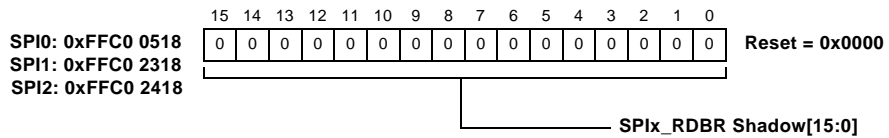


Figure 28-18. SPI RDBR Shadow Register

Programming Examples

This section includes examples ([Listing 28-1 on page 28-50](#) through [Listing 28-8 on page 28-57](#)) 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 28-1. SPI Register Initialization

```
SPIO_Register_Initialization:
    P0.H = hi(SPIO_FLG);
    P0.L = lo(SPIO_FLG);
    R0 = W[P0] (Z);
    BITSET (R0,0x3); /* FLS3 */
    W[P0] = R0; /* Enable slave-select output pin */

    P0.H = hi(SPIO_BAUD);
    P0.L = lo(SPIO_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 : Discard Incoming 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 : SC Toggles At START Of 1 Data Bit
 * CPOL [11]   = 1 : Active HIGH Serial Clock
 * MSTR [12]   = 1 : Device Is Master
 * WOM [13]    = 0 : Normal MOSI/MISO Data Output
 *              (No Open Drain)
 * SPE [14]    = 1 : SPI Module Is Enabled
 * [15]        = 0 : RESERVED
 *****/
PO.H = hi(SPI0_CTL) ;
PO.L = lo(SPI0_CTL) ;
RO = 0x5908;
W[P0] = RO.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.

Programming Examples

Listing 28-2. Initiate Transfer

```
Initiate_Transfer:
    PO.H = hi(SPIO_FLG);
    PO.L = lo(SPIO_FLG);
    RO = W[PO] (Z);
    BITCLR (RO,0xB); /* FLG3 */
    W[PO] = RO; /* Drive 0 on enabled slave-select pin */

    PO.H = hi(SPIO_TDBR); /* SPI Transmit Register */
    PO.L = lo(SPIO_TDBR);
    RO = W[P1++] (z); /* Get First Data To Be Transmitted And
Increment Pointer */
    W[PO] = RO; /* Write to SPIO_TDBR */

    PO.H = hi(SPIO_RDBR);
    PO.L = lo(SPIO_RDBR);
    RO = W[PO] (z); /* Dummy read of SPIO_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 SPIO_RDBR initiates the next transfer.

Listing 28-3. SPIO Interrupt Handler

```
SPIO_Interrupt_Handler:
Process_SPIO_Sample:
    PO.H = hi(SPIO_TDBR); /* SPIO transmit register */
    PO.L = lo(SPIO_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 28-4. 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.

Programming Examples

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 28-5. 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 28-6. SPI Initialization

```

SPI_Register_Initialization:
    P0.H = hi(SPIO_FLG);
    P0.L = lo(SPIO_FLG);
    R0 = W[P0] (Z);
    BITSET (R0,0x3); /* FLS3 */
    W[P0] = R0; /* Enable slave-select output pin */

    P1.H = hi(SPIO_BAUD);
    P1.L = lo(SPIO_BAUD);
    R0.L = 0x208E; /* Write to SPIO 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 : SC toggles at START of 1 data bit
* CPOL [11]   = 1 : Active HIGH serial clock
* MSTR [12]   = 1 : Device is master
* WOM [13]    = 0 : Normal MOSI/MISO data output
*              (no open drain)
* SPE [14]    = 1 : SPI module is enabled
*****/

```

Programming Examples

```
* [15]          = 0 : RESERVED
*****/
/* Configure SPI0 as MASTER */
R1 = 0x190B(z); /* Leave disabled until DMA is enabled*/
P1.L = 1o(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 28-7. Starting a Transfer

```
Initiate_Transfer:
    P0.H = hi(DMA16_4_CONFIG);
    P0.L = 1o(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 = 1o(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 28-8. 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(SPIO_STAT);
    P0.H = hi(SPIO_STAT);
    R1 = TXS; /* 0x08 */

Check_TXS: /* Poll for TXS = 0 */
    R2 = W[P0] (Z);
    R2 = R2 & R1;
    
```

Programming Examples

```
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 */
```

29 TWO WIRE INTERFACE CONTROLLERS

ADSP-BF54x 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 *I²C Bus Specification*, version 2.1, dated January 2000.

This chapter contains the following sections:

- “Overview” on page 29-1
- “Interface Overview” on page 29-3
- “Description of Operation” on page 29-7
- “TWI General Operation” on page 29-11
- “Functional Description” on page 29-22
- “Programming Model” on page 29-32
- “TWI Registers” on page 29-34
- “Programming Examples” on page 29-55
- “Electrical Specifications” on page 29-67

Overview

Each TWI is fully compatible with the widely used I²C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations.

Overview

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 I²C 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 the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*.

Interface Overview

Figure 29-1 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.

Interface Overview

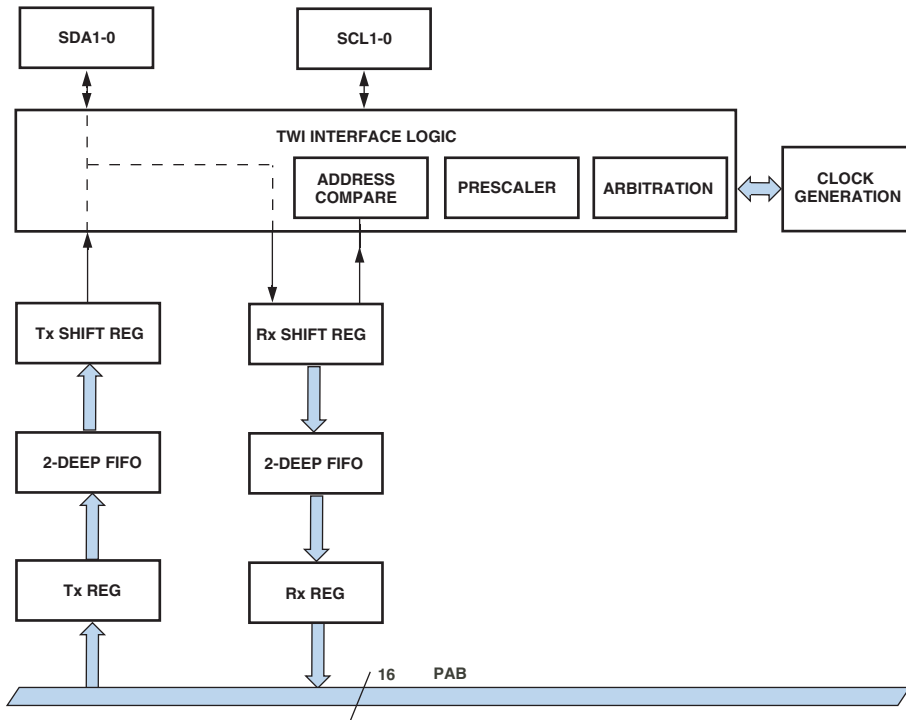


Figure 29-1. TWI Block Diagram

External Interface

The TWI signals are functionally multiplexed as general-purpose I/O. These signals, SDA_x (serial data) and SCL_x (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.

$$\text{PRESCALE} = f_{\text{SCLK}}/10\text{MHz}$$

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.

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 29-1 shows the pins for the TWI. Two bidirectional pins externally interface the TWI controller to the I²C bus. The interface is simple and no other external connections or logic are required.

Table 29-1. 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 a 1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event 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 I²C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in [Figure 29-2](#).

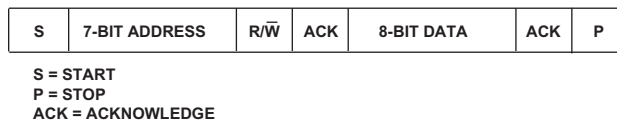


Figure 29-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, [Figure 29-3](#) 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.

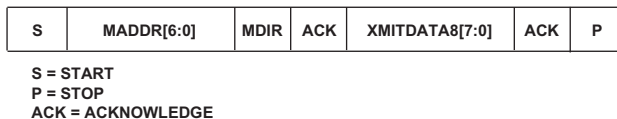


Figure 29-3. Data Transfer With Bit Illustration

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 mul-

Description of Operation

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 29-4](#) for TWI controller 0.

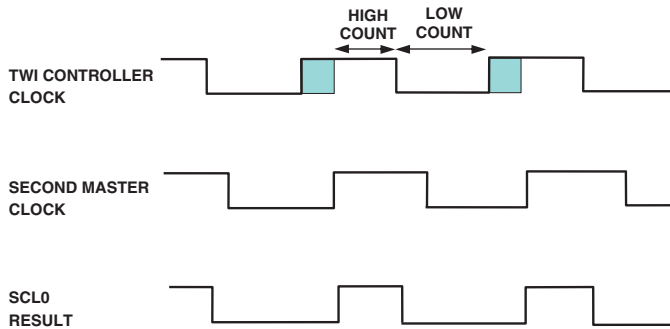


Figure 29-4. TWI Clock Synchronization

The TWI controller's serial clock (SCL_x) output follows these rules:

- Once the clock high ($CLKHI$) count is complete, the serial clock output is driven low and the clock low ($CLKLOW$) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL_x 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 29-5](#).

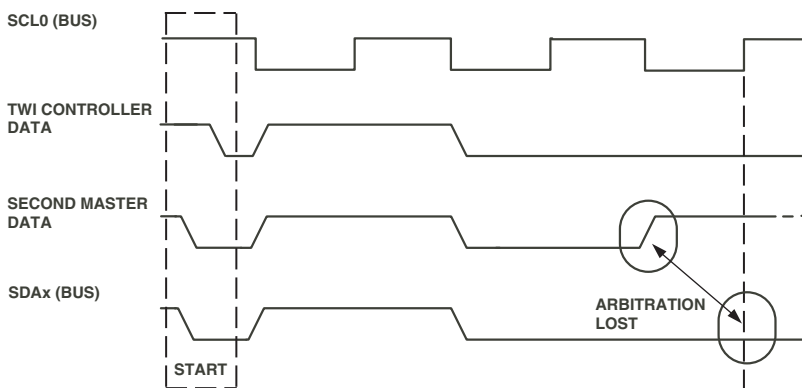


Figure 29-5. TWI Bus Arbitration

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 29-6](#).

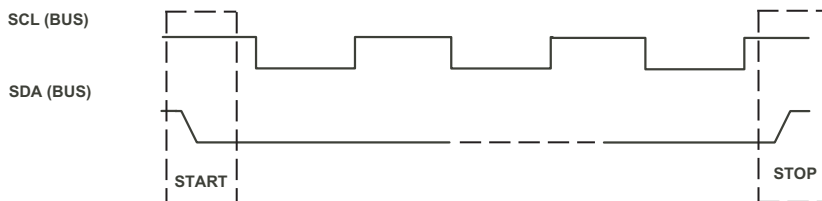


Figure 29-6. TWI Start and Stop Conditions

Description of Operation

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.

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.

$$\text{PRESCALE} = f_{\text{SCLK}}/10\text{MHz}$$

SCCB compatibility is an optional feature and should not be used in an I²C 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.

TWI General Operation

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 remain 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.

$$\text{CLKDIV} = \text{TWI } SCLx \text{ period} / 10 \text{ MHz time reference}$$

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):

$$\text{CLKDIV} = 2500 \text{ ns} / 100 \text{ ns} = 25$$

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.

Error Signals and Flags

The following sections describe the TWI error signals and flags.

TWI Master Status

The TWI master mode status register (`TWIX_MASTER_STAT`) holds information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

- **Bus busy** (`BUSBUSY`)

Indicates whether the bus is currently busy or free. This indication is not limited to only this device but is for all devices. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after t_{BUF} .

[1] The bus is busy. Clock or data activity 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.

TWI General Operation

[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 General Operation

TWI Slave Status

During and at the conclusion of slave mode transfers, the TWI slave mode status register (`TWIX_SLAVE_STAT`) holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

- **General call** (`GCALL`)

This bit self clears if slave mode is disabled (`SEN = 0`).

[1] At the time of addressing, the address was determined to be a general call.

[0] At the time of addressing, the address was not determined to be a general call.

- **Slave transfer direction** (`SDIR`)

This bit self clears if slave mode is disabled (`SEN = 0`).

[1] At the time of addressing, the transfer direction was determined to be slave transmit.

[0] At the time of addressing, the transfer direction was determined to be slave receive.

TWI FIFO Status

The fields in the TWI FIFO status register (`TWIX_FIFO_STAT`) indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

- **Receive FIFO status** (`RCVSTAT[1:0]`)

The `RCVSTAT` field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

[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.

TWI General Operation

[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 Status

The TWI interrupt status register (`TWIX_INT_STAT`) contains information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit by writing a 1 to it.

- **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 I²C bus transfers. It will be initially set on an I²C 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 I²C bus transfers. It will be initially set on an I²C 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 (RCV SERV)**

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 (XMT SERV)**

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)**

TWI General Operation

[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.

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 $f_{SCLK} / 10\text{MHz}$

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 29-2 shows what the interaction between the TWI controllers and the processor might look like using this example.

Table 29-2. Slave Mode Setup Interaction

TWI Controller	Processor
Interrupt: <code>SINIT</code> – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: <code>RCV SERV</code> – Receive buffer is full.	Acknowledge: Clear interrupt source bits. Read <code>TWIX_FIFO_STAT</code> . Read receive FIFO buffer.
...	...
Interrupt: <code>SCOMP</code> – 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 29-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 29-3. Master Mode Transmit Setup Interaction

TWI Controller	Processor
Interrupt: XMTSERV – Transmit buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
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.

Table 29-4 shows what the interaction between the TWI controllers and the processor might look like using this example.

Functional Description

Table 29-4. 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. This new behavior of the Master Mode operation utilizes self induced stretching of I²C clock while waiting on servicing interrupts.

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 will be 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 29-7](#) and described in [Table 29-5](#).

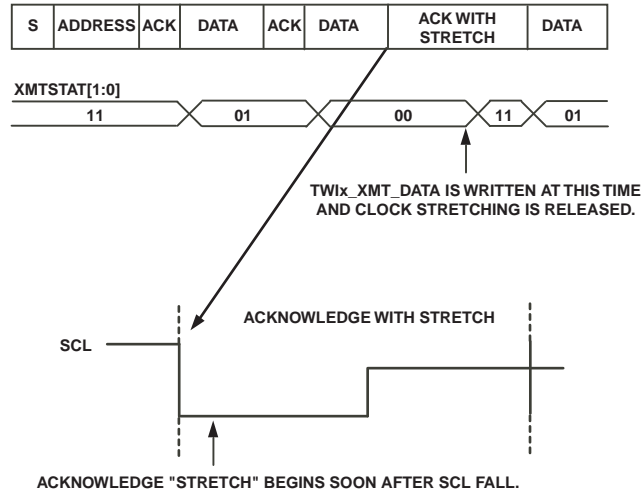


Figure 29-7. Clock Stretching during FIFO Underflow

Table 29-5. 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.

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (TWI_X_RCV_DATA8, TWI_X_RCV_DATA16). No other action is required to

Functional Description

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 29-8](#) and described in [Table 29-6](#).

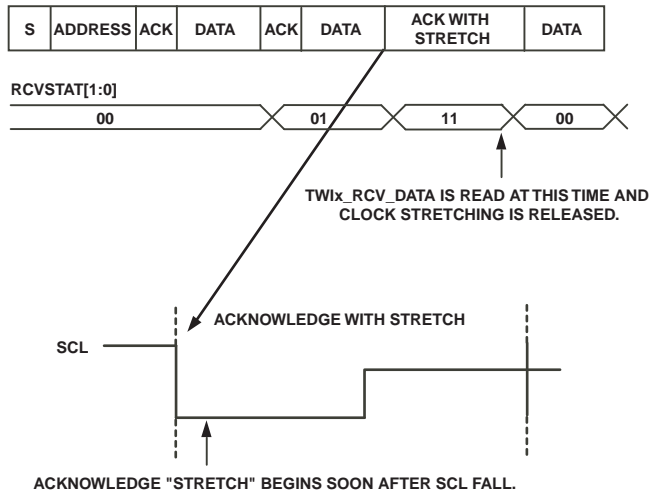


Figure 29-8. Clock Stretching During FIFO Overflow

Table 29-6. 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.

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. With the use of clock stretching the task of managing transitions becomes simpler and becomes common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock will initiate a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed ($DCNT = 0$). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in [Figure 29-9](#) and described in [Table 29-7](#).

Functional Description

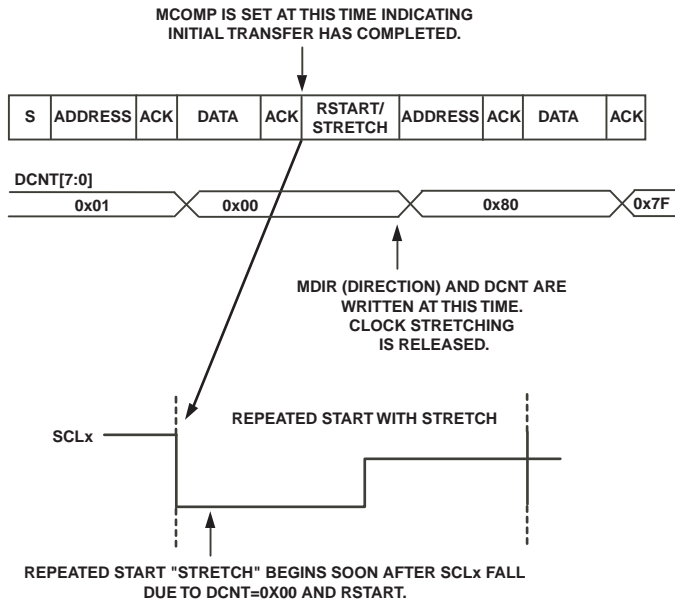


Figure 29-9. Clock Stretching during Repeated Start Condition

Table 29-7. 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.

Table 29-7. Repeated Start Case (Cont'd)

TWI Controller	Processor
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Programming Model

Figure 29-10 and Figure 29-11 illustrate the programming model for the TWI.

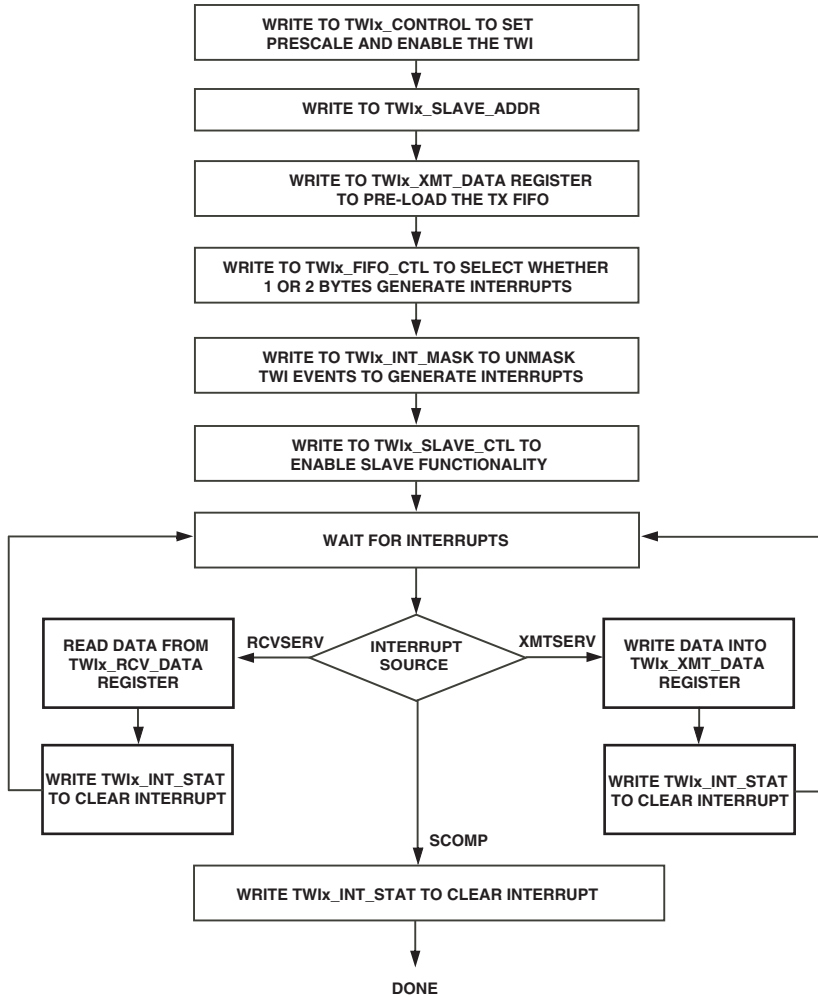


Figure 29-10. TWI Slave Mode

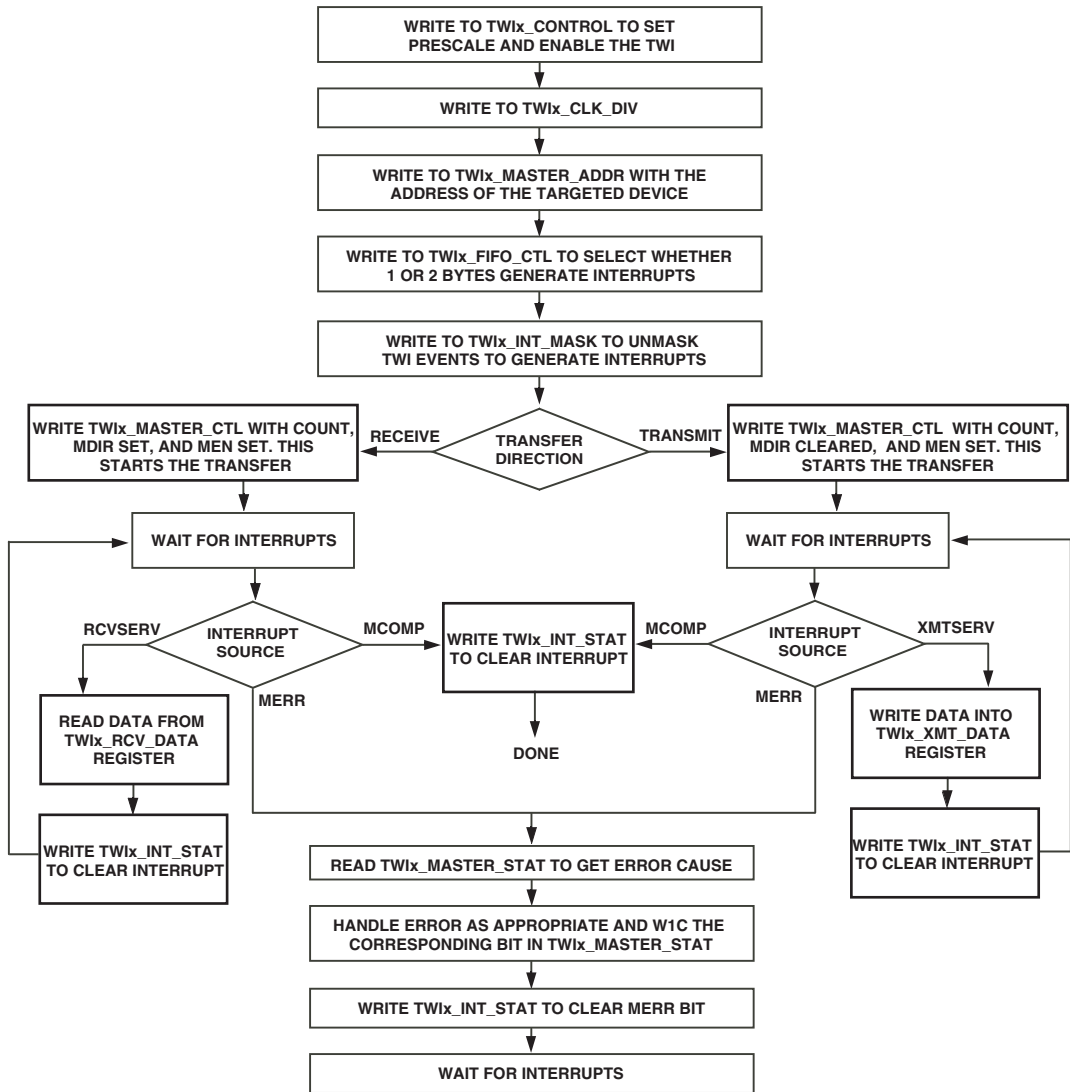


Figure 29-11. TWI Master Mode

TWI Registers

The TWI controller has 16 registers described in the following sections. [Table 29-8](#) lists the TWI registers.

Table 29-8. TWIx Registers

TWI0 Memory-mapped Registers	Register Name	Function
0xFFC0 0700	TWIX_CLKDIV	SCL clock divider registers on page 29-36
0xFFC0 0704	TWIX_CONTROL	TWI control registers on page 29-36
0xFFC0 0708	TWIX_SLAVE_CTL	TWI slave mode control registers on page 29-37
0xFFC0 070C	TWIX_SLAVE_ADDR	TWI slave mode address registers on page 29-39
0xFFC0 0710	TWIX_SLAVE_STAT	TWI slave mode status registers on page 29-40
0xFFC0 0714	TWIX_MASTER_CTL	TWI master mode control registers on page 29-41
0xFFC0 0718	TWIX_MASTER_ADDR	TWI master mode address registers on page 29-44
0xFFC0 071C	TWIX_MASTER_STAT	TWI master mode status registers on page 29-45
0xFFC0 0720	TWIX_INT_STAT	TWI interrupt status registers on page 29-51
0xFFC0 0724	TWIX_INT_MASK	TWI interrupt mask registers on page 29-47
0xFFC0 0728	TWIX_FIFO_CTL	TWI FIFO control registers on page 29-45
0xFFC0 072C	TWIX_FIFO_STAT	TWI FIFO status registers on page 29-47

Table 29-8. TWIx Registers (Cont'd)

TWI0 Memory-mapped Registers	Register Name	Function
0xFFC0 0780	TWIx_XMT_DATA8	TWI FIFO transmit data single-byte registers on page 29-52
0xFFC0 0784	TWIx_XMT_DATA16	TWI FIFO transmit data double-byte registers on page 29-52
0xFFC0 0788	TWIx_RCV_DATA8	TWI FIFO receive data single-byte registers on page 29-53
0xFFC0 078C	TWIx_RCV_DATA16	TWI FIFO receive data double-byte registers on page 29-54

TWI Registers

TWix_CONTROL Register

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.

TWI0_CONTROL Register

`TWI0_CONTROL 0xFFC00704`
`TWI1_CONTROL 0xFFC02204`

Reset = 0x0000

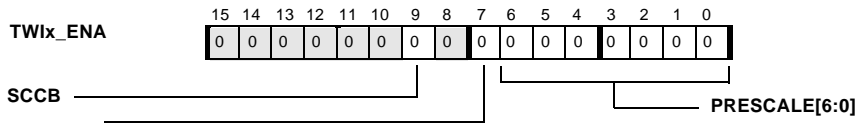


Figure 29-12. TWI Control Register

TWix_CLKDIV Register

During master mode operation, the `SCLx` clock divider register 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.

SCLx Clock Divider Register (TWIx_CLKDIV)

TWI0_CLKDIV 0xFFC00700
 TWI1_CLKDIV 0xFFC02200

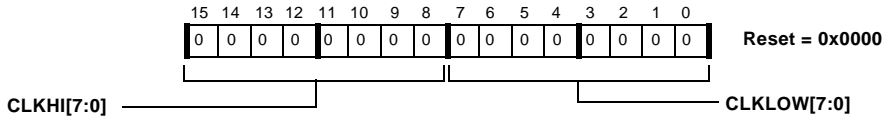


Figure 29-13. SCLx Clock Divider Register

TWIx_SLAVE_CTL Register

The TWI slave mode control register (TWIx_SLAVE_CTL) controls the logic associated with slave mode operation. Settings in this register do not affect master mode operation and should not be modified to control master mode functionality.

TWI Slave Mode Control Register (TWIx_SLAVE_CTL)

TWI0_SLAVE_CTL 0xFFC00708
 TWI1_SLAVE_CTL 0xFFC02208

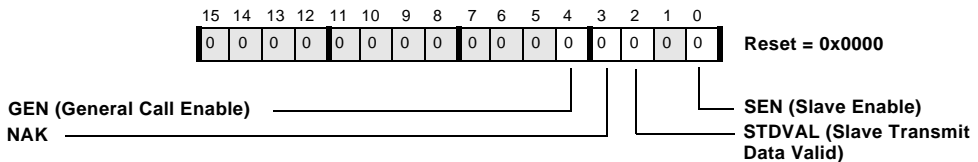


Figure 29-14. TWI Slave Mode Control Register

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.

TWI Registers

[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 will be NAK'ed (not acknowledged) back to the master yet the byte will be accepted into the receive FIFO and receive FIFO status (RCVSTAT) will be 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.

TWix_SLAVE_ADDR Register

The TWI slave mode address register (`TWix_SLAVE_ADDR`) holds the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

TWI Slave Mode Address Register (`TWix_SLAVE_ADDR`)

`TW10_SLAVE_ADDR 0xFFC00710`

`TW11_SLAVE_ADDR 0xFFC02210`

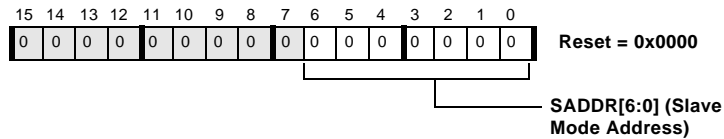


Figure 29-15. TWI Slave Mode Address Register

TWI Registers

TWIX_SLAVE_STAT Register

During and at the conclusion of slave mode transfers, the TWI slave mode status register (TWIX_SLAVE_STAT) holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

TWI Slave Mode Status Register (TWIX_SLAVE_STAT)

TWI0_SLAVE_STAT 0xFFC0070C

TWI1_SLAVE_STAT 0xFFC0220C

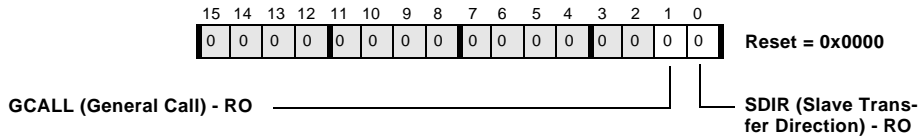


Figure 29-16. TWI Slave Mode Status Register

TWix_MASTER_CTL Register

The TWI master mode control register (TWIX_MASTER_CTL) controls the logic associated with master mode operation. Bits in this register do not affect slave mode operation and should not be modified to control slave mode functionality.

TWI Master Mode Control Register (TWIX_MASTER_CTL)

TW0_MASTER_CTL 0xFFC00714

TW1_MASTER_CTL 0xFFC02214

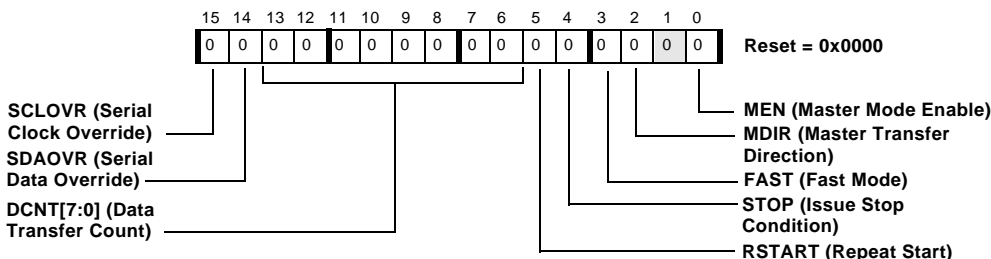


Figure 29-17. TWI Master Mode Control Register

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.

TWI Registers

- **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 will equal the number of bytes not sent. The byte which was NAK'ed by the slave will be 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. This includes transfers terminated due to errors.

[1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

[0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDAx, SCLx) are no longer driven. Write-1-to-clear status bits are not affected.

TWIX_MASTER_ADDR Register

During the transmit phase of a transfer, the TWI controllers, with their master enabled, transmits the contents of the TWI master mode address register (TWIX_MASTER_ADDR). When programming this register, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to this register. For example, if the slave address is $b\#1010000X$, where X is the read/write bit, then 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

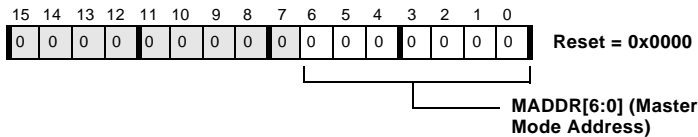


Figure 29-18. TWI Master Mode Address Register

TWix_MASTER_STAT Register

TWI Master Mode Status Register (TWix_MASTER_STAT)

TWI0_MASTER_STAT 0xFFC00718

TWI1_MASTER_STAT 0xFFC02218

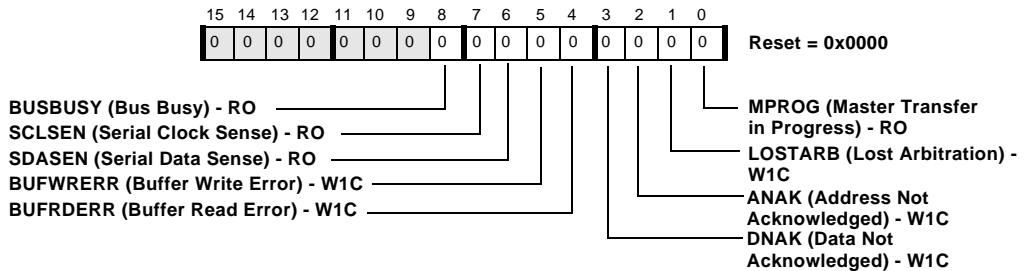


Figure 29-19. TWI Master Mode Status Register

TWix_FIFO_CTL Register

The TWI FIFO control register (TWix_FIFO_CTL) control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

TWI FIFO Control Register (TWix_FIFO_CTL)

TWI0_FIFO_CTL 0xFFC00728

TWI1_FIFO_CTL 0xFFC02228

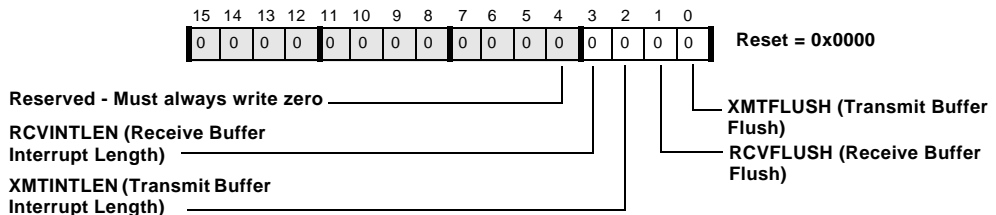


Figure 29-20. TWI FIFO Control Register

TWI Registers

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.

TWIX_FIFO_STAT Register

TWI FIFO Status Register (TWIX_FIFO_STAT)

All bits are RO.

TWIO_FIFO_STAT 0xFFC0072C

TWI1_FIFO_STAT 0xFFC0222C

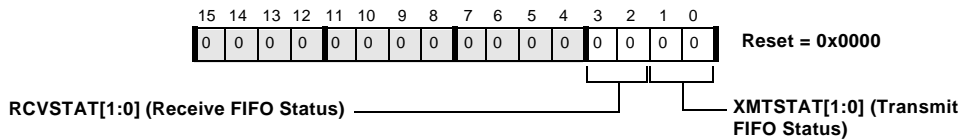


Figure 29-21. TWI FIFO Status Register

TWIX_INT_MASK Register

The TWI interrupt mask register (TWIX_INT_MASK) 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 Registers

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

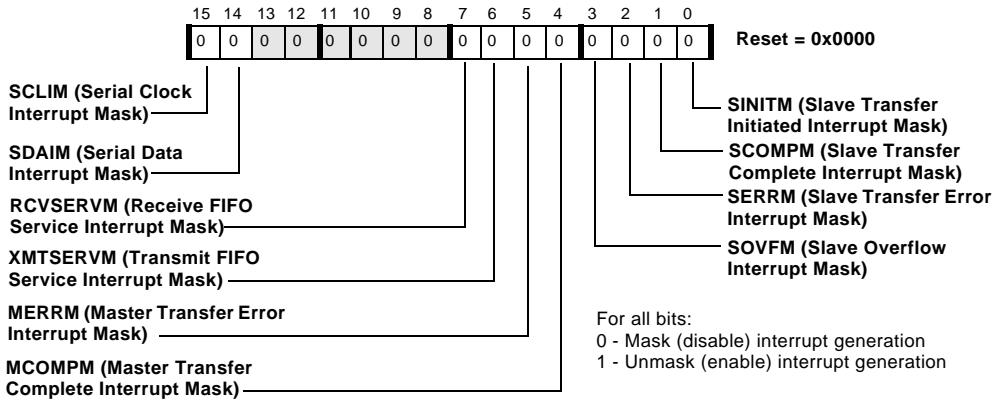


Figure 29-22. TWI Interrupt Mask Register

Additional information for the TWIx_INT_MASK register bits includes:

- **Serial Clock Interrupt Mask (SCLIM)**

[1] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

- **Serial Data Interrupt Mask (SDAIM)**

[1] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

- **Receive FIFO service interrupt mask** (RCVSERVM)

[1] The corresponding interrupt source is prevented from asserting the interrupt output.

[0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Transmit FIFO service interrupt mask** (XMTSERVM)

[1] The corresponding interrupt source is prevented from asserting the interrupt output.

[0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

TWI Registers

- **Master transfer error interrupt mask** (MERRM)
 - [1] The corresponding interrupt source is prevented from asserting the interrupt output.
 - [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.
- **Master transfer complete interrupt mask** (MCOMPMM)
 - [1] The corresponding interrupt source is prevented from asserting the interrupt output.
 - [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.
- **Slave overflow interrupt mask** (SOVFM)
 - [1] The corresponding interrupt source is prevented from asserting the interrupt output.
 - [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.
- **Slave transfer error interrupt mask** (SERRM)
 - [1] The corresponding interrupt source is prevented from asserting the interrupt output.
 - [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave transfer complete interrupt mask** (SCOMPM)

[1] The corresponding interrupt source is prevented from asserting the interrupt output.

[0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave transfer initiated interrupt mask** (SINITM)

[1] The corresponding interrupt source is prevented from asserting the interrupt output.

[0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

TWix_INT_STAT Register

TWI Interrupt Status Register (TWIx_INT_STAT)

All bits are sticky and W1C.

TWIO_INT_STAT 0xFFC00720

TWI1_INT_STAT 0xFFC02220

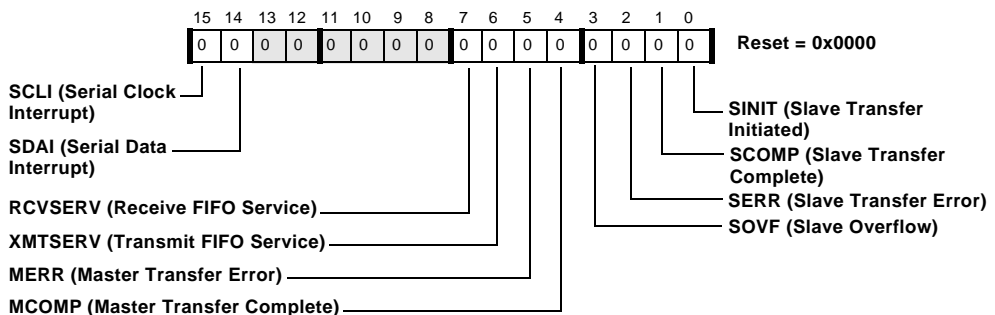


Figure 29-23. TWI Interrupt Status Register

TWIX_XMT_DATA8 Register

The TWI FIFO transmit data single byte register (TWIX_XMT_DATA8) holds an 8-bit data value written into the FIFO buffer. Transmit data is entered into the corresponding transmit buffer in a first-in first-out order.

Although peripheral bus writes are 16 bits, a write access to 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.

TWIO_XMT_DATA8 0xFFC00780
 TWI1_XMT_DATA8 0xFFC02280

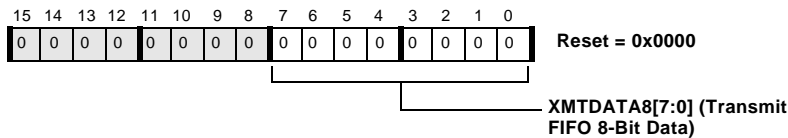


Figure 29-24. TWI FIFO Transmit Data Single Byte Register

TWIX_XMT_DATA16 Register

The TWI FIFO transmit data double byte register (TWIX_XMT_DATA16) holds a 16-bit data value written into the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access. The data is written in little endian byte order as shown in Figure 29-25 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 29-25. Little Endian Byte Order

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

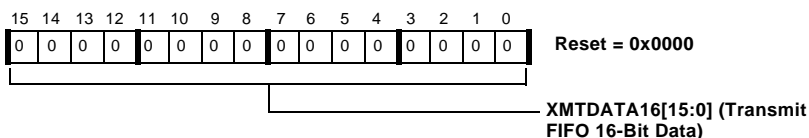


Figure 29-26. TWI FIFO Transmit Data Double Byte Register

TWIX_RCV_DATA8 Register

The TWI FIFO receive data single byte register (TWIX_RCV_DATA8) holds an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWIX_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWIX_FIFO_STAT 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 Registers

TWI FIFO Receive Data Single Byte Register (TWIx_RCV_DATA8)

All bits are RO.

TWI0_RCV_DATA8 0xFFC00788
TWI1_RCV_DATA8 0xFFC02288

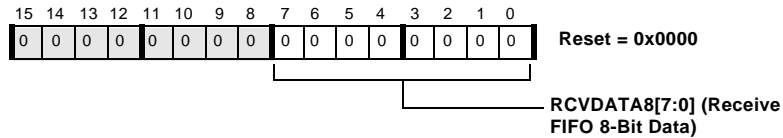


Figure 29-27. TWI FIFO Receive Data Single Byte Register

TWix_RCV_DATA16 Register

The TWI FIFO receive data double byte register (TWIx_RCV_DATA16) holds a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access. The data is read in little endian byte order as shown in [Figure 29-28](#) 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 29-28. Little Endian Byte Order

TWI FIFO Receive Data Double Byte Register (TWIx_RCV_DATA16)

All bits are WO.

TWI0_RCV_DATA16 0xFFC0078C
 TWI1_RCV_DATA16 0xFFC0228C

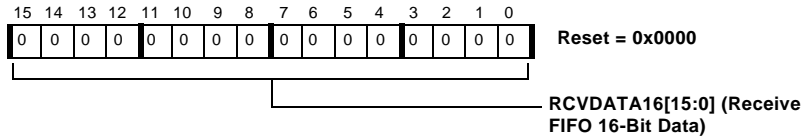


Figure 29-29. TWI FIFO Receive Data Double Byte Register

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

Master Mode Setup

[Listing 29-1](#) shows how to initiate polled receive and transmit transfers in master mode.

Listing 29-1. 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";
    
```

Programming Examples

```
.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
```

Two Wire Interface Controllers

```

*****/
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWIO_INT_MASK)] = R1;

/*****
The address needs to be shifted one place to the right
e.g., 1010 001x becomes 0101 0001 (0x51) the TWIO controller
will actually send out 1010 001x where x is either a 0 for
writes or 1 for reads
*****/
R6 = 0xBF;
R6 = R6 >> 1;
TWIO_INIT.END: W[P1 + LO(TWIO_MASTER_ADDR)] = R6;

/***** END OF TWIO INIT *****/

/*****
Starting the Read transfer
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or SLOW
4. direction of transfer:
    MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. This will kick off the master transfer
*****/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWIO_MASTER_CTL)] = R1;
ssync;

/*****
Poll the FIFO Status register to know when
2 bytes have been shifted into the RX FIFO
*****/

```

Programming Examples

```
Rx_stat:
R1 = W[P1 + LO(TWIO_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF !cc jump Rx_stat;
R0 = W[P1 + LO(TWIO_RCV_DATA16)](Z); /* Read data from the RX
fifo */
ssync;

/*****
check that master transfer has completed
MCOMP will be set when Count reaches zero
*****/
M_COMP:
    R1 = W[P1 + LO(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 will kick 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) LCO = 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
```

```
*****/
```

Programming Examples

```
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 29-2](#) shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine `_TWIO_ISR` shown in [Listing 29-3](#).

Listing 29-2. 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];
```

Two Wire Interface Controllers

```
.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 will respond 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);
```

Programming Examples

```
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(TWIO_FIFO_CTL)] = R1;
```

```
/******  
enable these signals to generate a TWIO interrupt
```

```
*****/
```

```
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
```

```
W[P1 + LO(TWIO_INT_MASK)] = R1;
```

```
/******
```

```
Enable the TWIO 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(TWIO_SLAVE_CTL)] = R1;
```

```
TWIO_SLAVE_INIT.END:
```

```
P2.H = HI(TWIO_RX);
```

```
P2.L = LO(TWIO_RX);
```

```
P4.H = HI(TWIO_TX);
```

```
P4.L = LO(TWIO_TX);
```

```

/*****
Remap the vector table pointer from the default __I10HANDLER
to the new _TWIO_ISR interrupt service routine
*****/
R1.H = HI(_TWIO_ISR);
R1.L = LO(_TWIO_ISR);
[P0 + LO(EVT10)] = R1; /* note that P0 points to the base of the
core MMR registers */

/*****
ENABLE TWIO generate to interrupts at the system level
*****/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWIO));
[P1 + LO(SIC_IMASK)] = R1;

/*****
ENABLE TWIO 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:

```

Programming Examples

Listing 29-3. TWIO Slave Interrupt Service Routine

```

/*****
Function:  _TWIO_ISR
Description:  This ISR is executed when the TWIO controller
detects a slave initiated transfer. After an interrupt is ser-
viced, its corresponding bit is cleared in the TWIO_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(TWIO_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWIO_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWIO_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(TWIO_INT_STAT)] = R0; /*
clear interrupt source bit */
ssync;

```

Programming Examples

```
JUMP _TWIO_ISR.END; /* exit */

/*****
slave overflow
*****/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWIO_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWIO_ISR.END; /* exit */

/*****
slave transfer complete
*****/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWIO_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWIO_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc. ... */
R0 = W[P1 + LO(TWIO_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 _TWIO_ISR.END;
R0 = W[P1 + LO(TWIO_RCV_DATA8)](Z); /* read data */
B[P2++] = R0 ; /* store bytes into a buffer pointed to by P2 */

_TWIO_ISR.END:RTI;
```


Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I²C Bus Specification version 2.1* dated January 2000.

Electrical Specifications

30 SPORT CONTROLLERS

This chapter describes the processor's dual-channel synchronous serial ports (SPORTs) and includes the following sections:

- [“Overview” on page 30-1](#)
- [“Interface Overview” on page 30-3](#)
- [“Description of Operation” on page 30-11](#)
- [“Functional Description” on page 30-28](#)
- [“SPORT Registers” on page 30-48](#)
- [“Programming Examples” on page 30-76](#)

Overview

The ADSP-BF54x 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 I²S Mode
- H.100 Telephony standard support

Overview

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.

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 I²S 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 30-31](#) 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 on the port configuration, see the “General Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference*. SPORTs provide synchronous serial

Interface Overview

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 RFS_x signals indicates RFS_0 , RFS_1 , RFS_2 , and RFS_3 (corresponding to $SPORT_0$, $SPORT_1$, $SPORT_2$, and $SPORT_3$, respectively). In this chapter, LSB refers to least significant bit, and MSB refers to most significant bit.

Port A contains the $SPORT_2$ and $SPORT_3$ pins. Some of the $SPORT_2$ and $SPORT_3$ pins are multiplexed and can be used for other purposes if the entire $SPORT_2$ and $SPORT_3$ blocks or some of their signals are not required by an application. However, all pins default to the $SPORT_2$ and $SPORT_3$ modules settings after reset.

$SPORT_0$ resides in port C. Its secondary data pins are shared with $MXVR$. The $PORTC_MUX$ register controls whether the secondary $SPORT_0$ 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 the “General Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

$SPORT_1$ resides in port D. Its signals are shared with the PPI and HDMA. The $PORTD_MUX$ register controls whether the $SPORT_1$ 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 the “General Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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 the “General Purpose Ports” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference*.


 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 30-1 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 30-1 shows the signals for each SPORT.

Table 30-1. SPORTx Signals

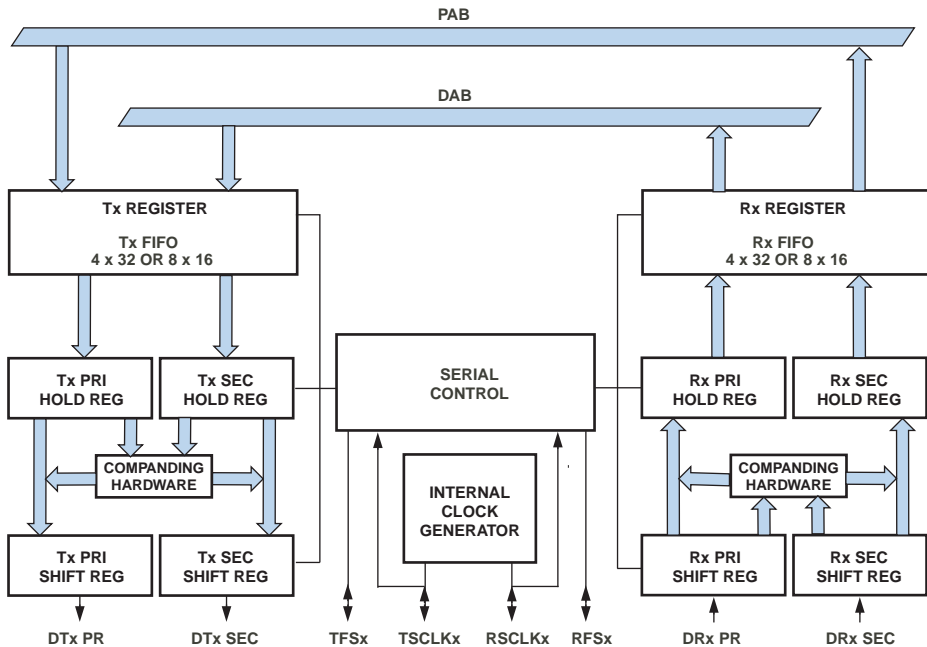
Pin ¹	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary

Interface Overview

Table 30-1. SPORT_x Signals (Cont'd)

Pin ¹	Description
RSCLK _x	Receive Clock
RFS _x	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).



NOTE 1: ALL WIDE ARROW DATA PATHS ARE 16 OR 32 BITS WIDE, DEPENDING ON SLEN. FOR SLEN = 2 TO 15, A 16-BIT DATA PATH WITH 8-DEEP FIFO IS USED. FOR SLEN = 16 TO 31, A 32-BIT DATA PATH WITH 4-DEEP FIFO IS USED.

NOTE 2: Tx REGISTER IS THE BOTTOM OF THE Tx FIFO, Rx REGISTER IS THE TOP OF THE Rx FIFO.

Figure 30-1. SPORT Block Diagram

A SPORT receives serial data on its DRxPRI and DRxSEC inputs and transmits serial data on its DTxPRI and DTxSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTxPRI and DTxSEC) are synchronous to the transmit clock (TSCLKx). For receive, the data bits (DRxPRI and DRxSEC) are synchronous to the receive clock (RSCLKx). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFSx and TFSx are used to indicate the start of a serial data word or stream of serial words.

Interface Overview

The primary and secondary data pins, if enabled by the port configuration, provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary signals is interleaved in main memory and can be retrieved by setting a stride in the Data Address Generators (DAG) unit. For more information about DAGs, see the “Data Address Generators” chapter in the *Blackfin Processor Programming Reference*. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor’s powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

[Figure 30-2](#) shows a possible port connection for the SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial 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 30-17](#) for details.

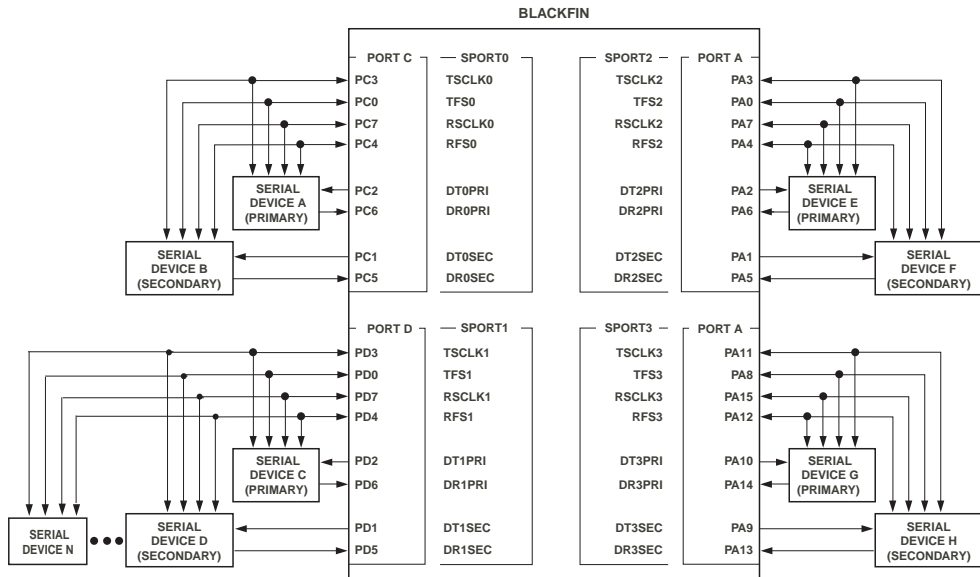


Figure 30-2. SPORT3-0 Example Connections

Interface Overview

Figure 30-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

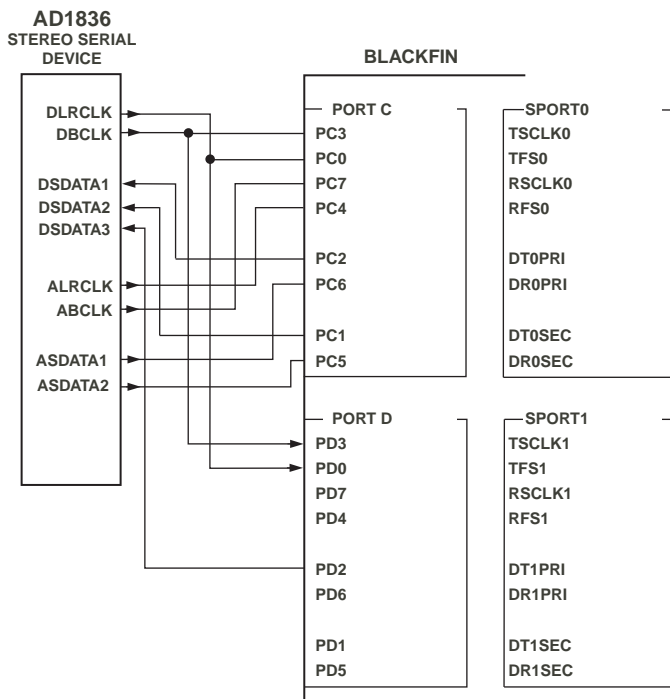


Figure 30-3. SPORT1–0 Example Stereo Serial Connection

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

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 information about DMA, see the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference*.


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.

Description of Operation

A processor reset disables the SPORTs by clearing the `SPORTx_TCR1`, `SPORTx_TCR2`, `SPORTx_RCR1`, and `SPORTx_RCR2` registers (including the `TSPEN` and `RSPEN` enable bits) and the `SPORTx_TCLKDIV`, `SPORTx_RCLKDIV`, `SPORTx_TFSDIVx`, and `SPORTx_RFSDIVx` clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the `TSPEN` and `RSPEN` enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

 Note that disabling a SPORT 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.

The SPORTs are ready to start transmitting or receiving data no later than three serial clock cycles after they are enabled in the `SPORTx_TCR1` or `SPORTx_RCR1` register. No serial clock cycles are lost from this point on. The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

When disabling the SPORT from multichannel operation, first disable `TXEN` and then disable `RXEN`. Note both `TXEN` and `RXEN` must be disabled before 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 I²S 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 I²S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special `LRCLK`-style frame sync. All other SPORT control bits remain in effect and should be set appropriately.

[Figure 30-4 on page 30-14](#) shows timing diagrams for stereo serial mode transmit operation.

[Figure 30-5 on page 30-15](#) shows timing diagrams for stereo serial mode receive operation.

 Blackfin SPORTs are designed such that, in I²S 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 I²S 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 I²S receiver is a Blackfin SPORT.

[Table 30-2](#) 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-

Description of Operation

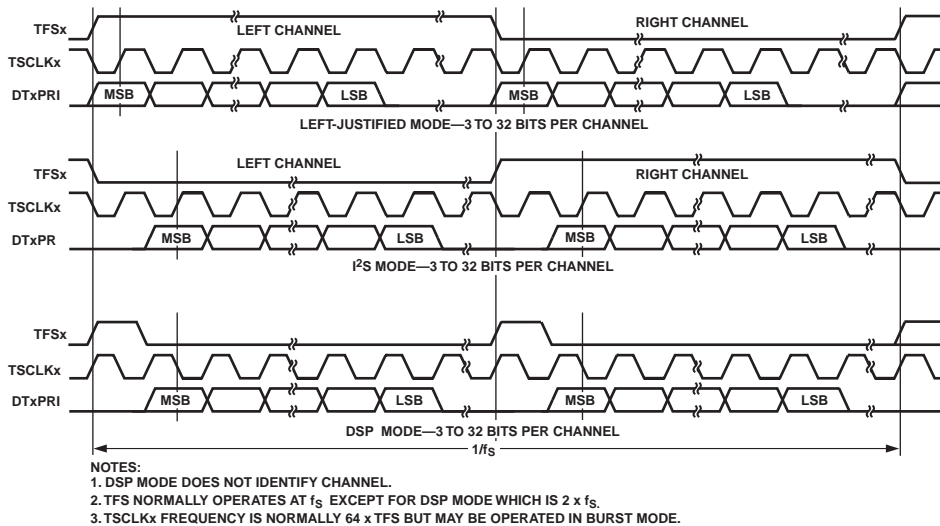


Figure 30-4. SPORT Stereo Serial Modes, Transmit

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."

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 30-2](#) 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 30-28) still apply,

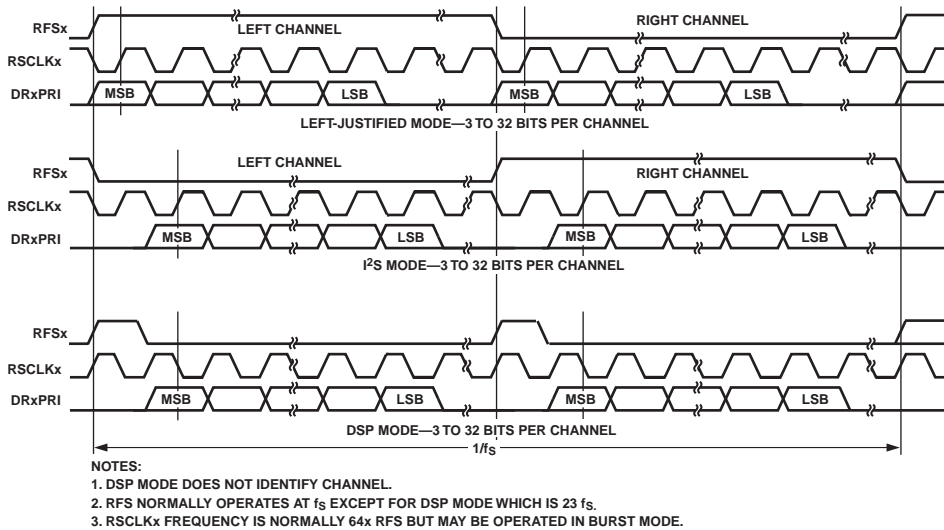


Figure 30-5. SPORT Stereo Serial Modes, Receive

Table 30-2. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I ² S	Left-Justified	DSP Mode
RFSFE	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

Description of Operation

Table 30-2. Stereo Serial Settings (Cont'd)

Bit Field	Stereo Audio Serial Scheme		
	I ² S	Left-Justified	DSP Mode
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

but now refer to one half the period and twice the frequency. For instance, setting $RFSDIV$ or $TFSDIV = 31$ produces an $LRCLK$ that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The $LRFS$ bit determines the polarity of the frame sync pin that is considered a “right” channel. Thus, setting $LRFS = 0$ indicates that a high signal on the $RFSx$ or $TFSx$ pin is the right channel, and a low signal is the left channel. This is the default setting.

The $RRFST$ and $TRFST$ bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary $DRxSEC$ and $DTxSEC$ pins are useful extensions of the SPORT which pair well with stereo serial mode. Multiple 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 30-3 on page 30-10](#), 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

Description of Operation

(not three-stated) if the SPORT is enabled and the secondary transmit is enabled ($TXSE = 1$ in the $SPORTx_TCR2$ register), unless the SPORT is in multichannel mode and an inactive time slot occurs.


 The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling $SPORTx_TX$ or $SPORTx_RX$ operation for multichannel mode. This is especially important in “DMA data unpacked mode,” since SPORT FIFO operation begins immediately after $RSPEN$ and $TSPEN$ are set, enabling both RX and TX. The $MCMEN$ bit (in $SPORTx_MCMC2$) must be enabled prior to enabling $SPORTx_TX$ or $SPORTx_RX$ operation. When disabling the SPORT from multichannel operation, first disable $TXEN$ and then disable $RXEN$. Note both $TXEN$ and $RXEN$ must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 30-6 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 30-41 for more examples.

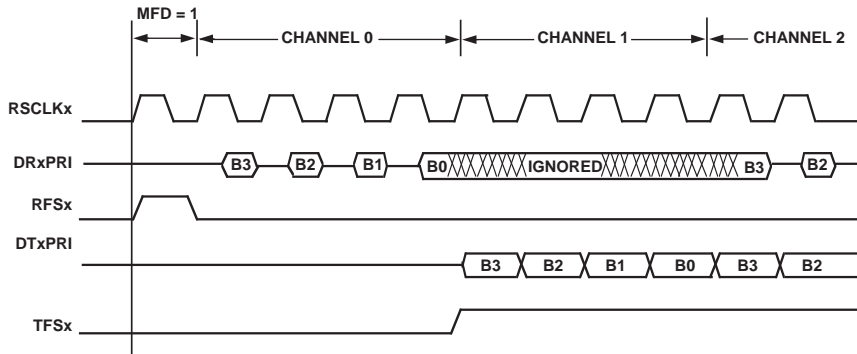


Figure 30-6. Multichannel Operation

Multichannel Enable

Setting the MCMEN bit in the SPORT_x_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

i 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.

Description of Operation

Table 30-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 30-3. Multichannel Mode Configuration

SPORT _x _RCR1 or SPORT _x _RCR2	SPORT _x _TCR1 or SPORT _x _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 RFS_x 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.

Description of Operation

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 30-7](#).

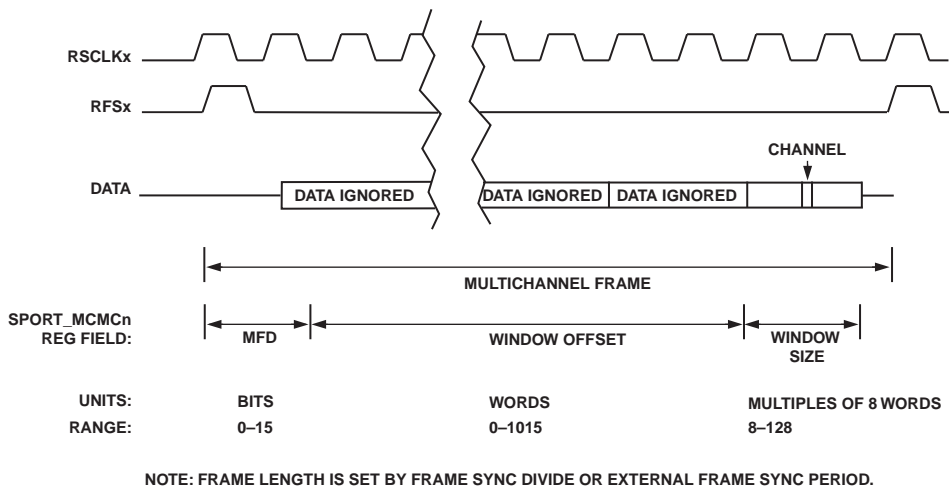


Figure 30-7. Relationships for Multichannel Parameters

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:

$$\text{Number of words in active window} = 8 \times (\text{WSIZE} + 1)$$

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Description of Operation

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 30-8](#).

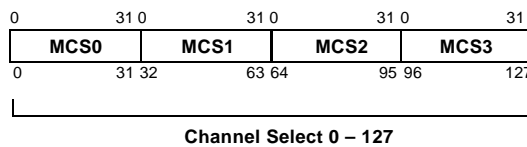


Figure 30-8. Multichannel Select Registers

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.

Description of Operation

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 30-31 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 = $(SCLK \text{ frequency}) / (2 \times (SPORTx_TCLKDIV + 1))$

$RSCLKx$ frequency = $(SCLK \text{ frequency}) / (2 \times (SPORTx_RCLKDIV + 1))$

If the value of $SPORTx_TCLKDIV$ or $SPORTx_RCLKDIV$ is changed while the internal serial clock is enabled, the change in $TSCLKx$ 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:

$$\text{SPORTxTFS frequency} = (\text{TSCLKx frequency}) / (\text{SPORTx_TFSDIV} + 1)$$

$$\text{SPORTxRFS frequency} = (\text{RSCLKx frequency}) / (\text{SPORTx_RFSDIV} + 1)$$

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORTx_TCR2 or SPORTx_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the product data sheet for exact timing specifications.


Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The

Functional Description

value of the serial word length (SLEN) field in the SPORT_x_TCR2 and SPORT_x_RCR2 registers of each SPORT determines the word length according to this formula:

$$\text{Serial Word Length} = \text{SLEN} + 1$$

 The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so 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 SPORT_x_RCR1 and SPORT_x_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 SPORT_x_TCR1 register and the RDTYPE field of the SPORT_x_RCR1 register specify one of four data formats for both single and multichannel operation. See [Table 30-4](#).

Table 30-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORT _x _TCR1 Data Formatting	SPORT _x _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.

Functional Description

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`.

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 30-9 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 30-41 for more timing examples.

Functional Description

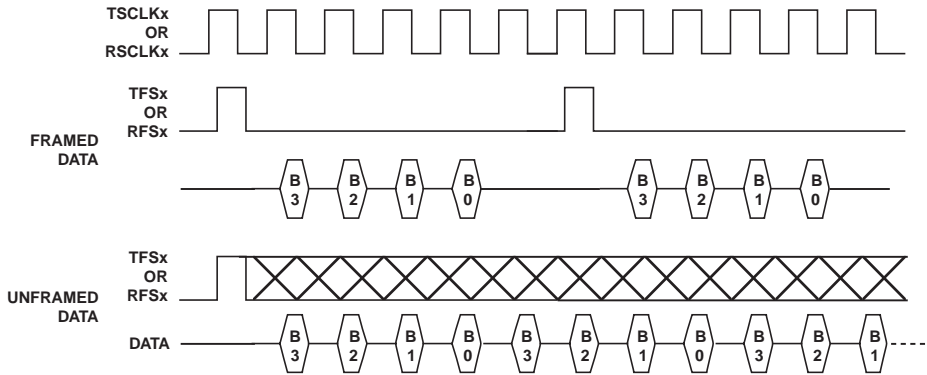


Figure 30-9. Framed Versus Unframed Data

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.

Functional Description

⚡ 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 30-10](#), $TCKFE = RCKFE = 0$ and transmit and receive are connected together to share the same clock and frame syncs.

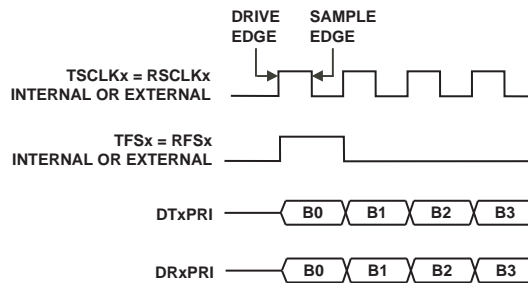


Figure 30-10. Example of $TCKFE = RCKFE = 0$, Transmit and Receive Connected

In [Figure 30-11](#), $TCKFE = RCKFE = 1$ and transmit and receive are connected together to share the same clock and frame syncs.

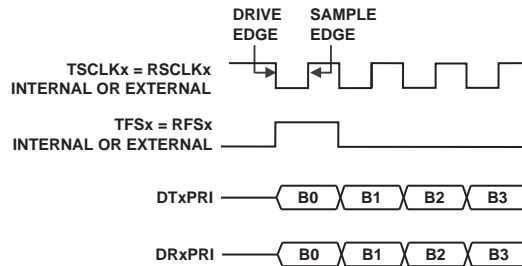


Figure 30-11. Example of $TCKFE = RCKFE = 1$, Transmit and Receive Connected

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 ≥ 3`).

Functional Description

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 30-12 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 30-41 for more examples.

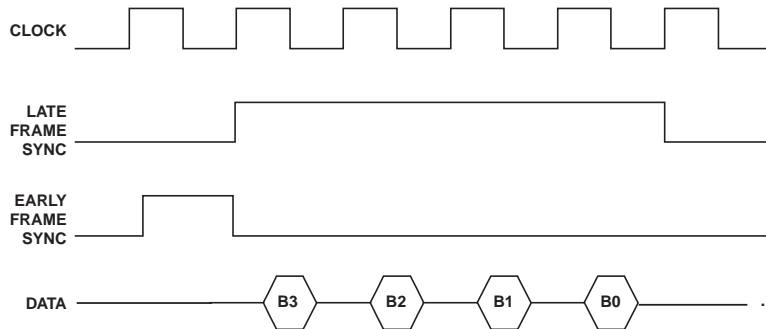


Figure 30-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS_x) is output only when the $SPORT_x_TX$ buffer has data ready to transmit. The data-independent transmit frame sync select bit ($DITFS$) allows the continuous generation of the TFS_x signal, with or without new data. The $DITFS$ bit of the $SPORT_x_TCR1$ register configures this option.

When $DITFS = 0$, the internally generated TFS_x is only output when a new data word is loaded into the $SPORT_x_TX$ buffer. The next TFS_x is generated once data is loaded into $SPORT_x_TX$. This mode of operation allows data to be transmitted only when it is available.

When $DITFS = 1$, the internally generated TFS_x is output at its programmed interval regardless of whether new data is available in the $SPORT_x_TX$ buffer. Whatever data is present in $SPORT_x_TX$ is transmitted again with each assertion of TFS_x . The $TUVF$ (transmit underflow status) bit in the $SPORT_x_STAT$ register is set when this occurs and old data is retransmitted. The $TUVF$ status bit is also set if the $SPORT_x_TX$ buffer does not have new data when an externally generated TFS_x occurs. Note that in this mode of operation, data is transmitted only at specified times.

Functional Description

If the internally generated TFS_x is used, a single write to the $SPORT_x_TX$ data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when $RSPEN$ is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when $TSPEN$ is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits ($ROVF$, $RUVF$, $TOVF$, $TUVF$) are set. The $ROVF$ and $RUVF$ bits are cleared by writing 0 to $RSPEN$. The $TOVF$ and $TUVF$ bits are cleared by writing 0 to $TSPEN$.

PAB Errors

The SPORT generates a PAB error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_x_TX)
- Writing a read-only register (for example, SPORT_x_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 30-32, “[Early Versus Late Frame Syncs \(Normal Versus Alternate Timing\)](#)” on page 30-37, and “[Frame Syncs in Multichannel Mode](#)” on page 30-20). 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 product data sheet for actual timing parameters and values.

These examples assume a word length of four bits ($SLEN = 3$). Framing signals are active high ($LRFS = 0$ and $LTFS = 0$).

[Figure 30-13](#) through [Figure 30-18](#) show framing for receiving data.

Functional Description

In [Figure 30-13](#) and [Figure 30-14](#), 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 $RSCLKx$ cycles between words).

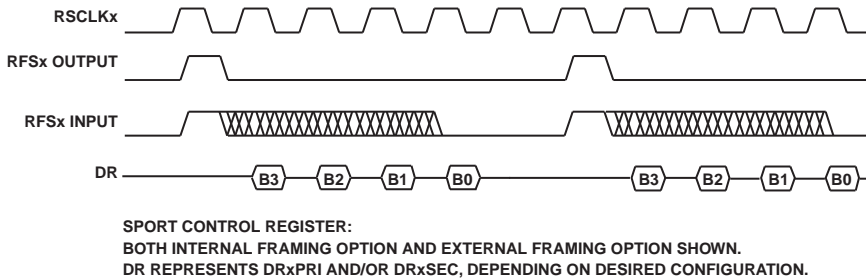


Figure 30-13. SPORT Receive, Normal Framing

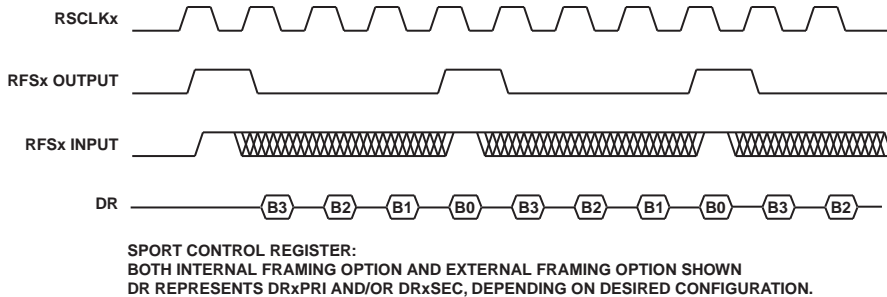


Figure 30-14. SPORT Continuous Receive, Normal Framing

Figure 30-15 and Figure 30-16 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.

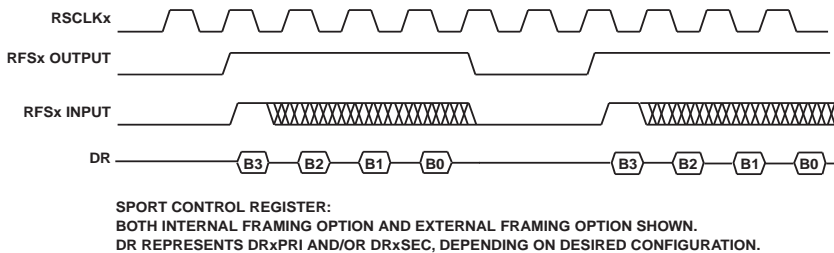


Figure 30-15. SPORT Receive, Alternate Framing

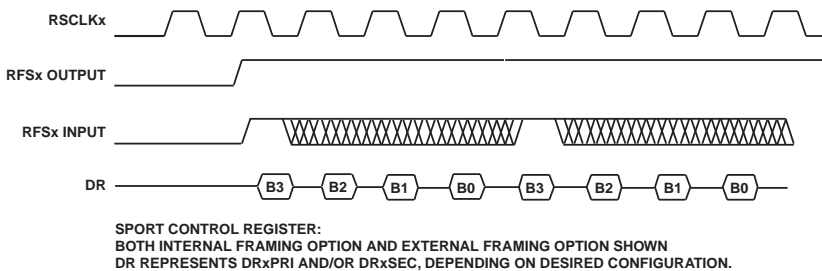


Figure 30-16. SPORT Continuous Receive, Alternate Framing

Functional Description

Figure 30-17 and Figure 30-18 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).

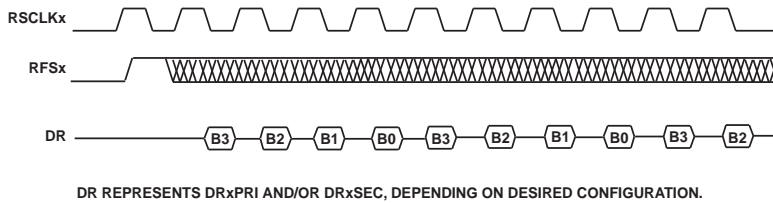


Figure 30-17. SPORT Receive, Unframed Mode, Normal Framing

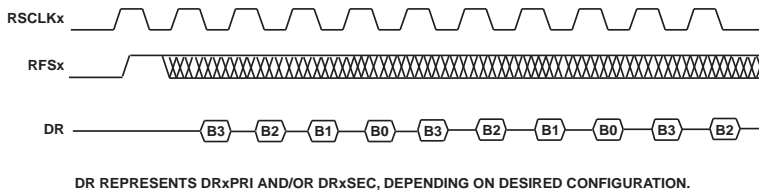


Figure 30-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 30-19 through Figure 30-24 show framing for transmitting data and are very similar to Figure 30-13 through Figure 30-18.

In Figure 30-19 and Figure 30-20, the normal framing mode is shown for non-continuous data (any number of $TSCLK_x$ cycles between words) and continuous data (no $TSCLK_x$ cycles between words). Figure 30-21 and Figure 30-22 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFS_x output meets the RFS_x input timing requirement.

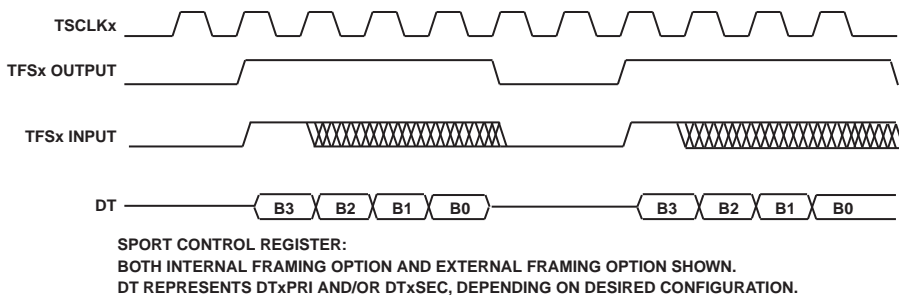


Figure 30-19. SPORT Transmit, Normal Framing

Functional Description

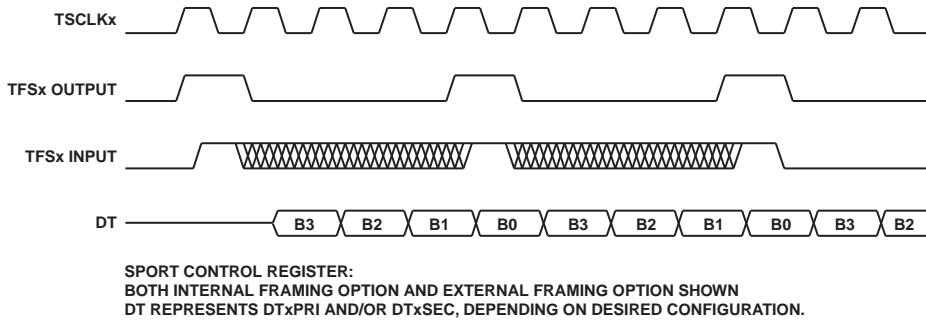


Figure 30-20. SPORT Continuous Transmit, Normal Framing

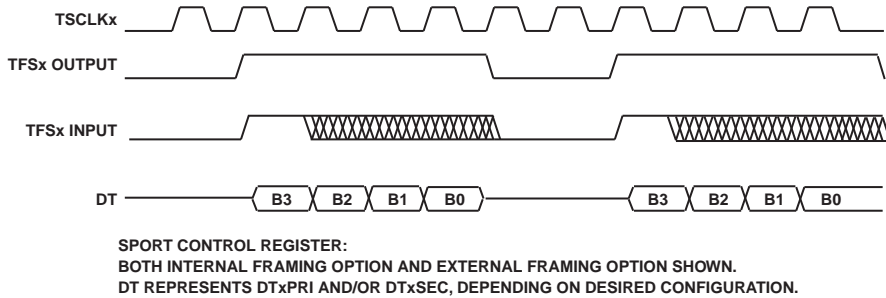


Figure 30-21. SPORT Transmit, Alternate Framing

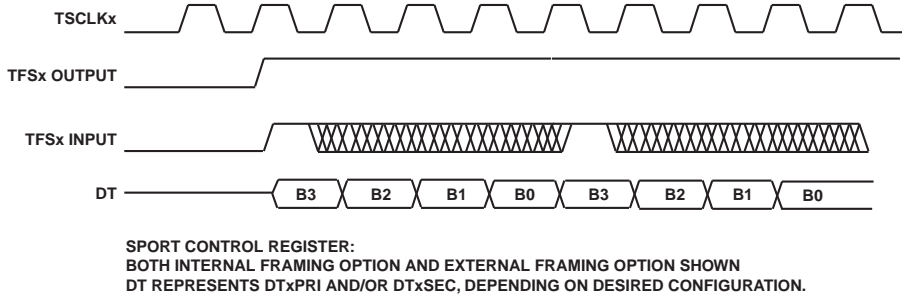


Figure 30-22. SPORT Continuous Transmit, Alternate Framing

Figure 30-23 and Figure 30-24 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 TCLKx before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

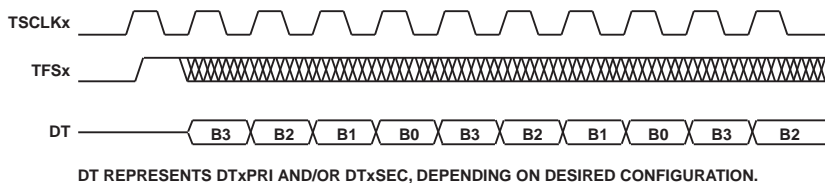


Figure 30-23. SPORT Transmit, Unframed Mode, Normal Framing

SPORT Registers

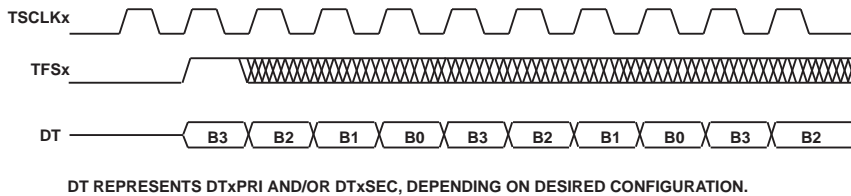


Figure 30-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. [Table 30-5](#) provides an overview of the available control registers.

Table 30-5. SPORT Registers

Register Name	Function	Notes
SPORTx_TCR1	Primary transmit configuration registers on page 30-51	Bits [15:1] can only be written if bit 0 = 0
SPORTx_TCR2	Secondary transmit configuration registers on page 30-51	
SPORTx_TCLKDIV	Transmit clock divider registers on page 30-68	Ignored if external SPORT clock mode is selected
SPORTx_TFSDIV	Transmit frame sync divider registers on page 30-69	Ignored if external frame sync mode is selected
SPORTx_TX	SPORT transmit data registers on page 30-61	See description of FIFO buffering at “SPORTx_TX Register” on page 30-61

Table 30-5. SPORT Registers (Cont'd)

Register Name	Function	Notes
SPORTx_RCR1	Primary receive configuration registers on page 30-56	Bits [15:1] can only be written if bit 0 = 0
SPORTx_RCR2	Secondary receive configuration registers on page 30-56	
SPORTx_RCLKDIV	Receive clock divider registers on page 30-68	Ignored if external SPORT clock mode is selected
SPORTx_RFSDIV	Receive frame sync divider registers on page 30-69	Ignored if external frame sync mode is selected
SPORTx_RX	SPORT receive data registers on page 30-63	See description of FIFO buffering at “SPORTx_RX Register” on page 30-63
SPORTx_STAT	Receive and transmit status registers on page 30-66	
SPORTx_MCMC1	Primary multichannel mode configuration registers on page 30-70	Configure this register before enabling the SPORT
SPORTx_MCMC2	Secondary multichannel mode configuration registers on page 30-70	Configure this register before enabling the SPORT
SPORTx_MRCSn	Receive channel selection registers on page 30-72	Select or deselect channels in a multichannel frame
SPORTx_MTCSn	Transmit channel selection registers on page 30-74	Select or deselect channels in a multichannel frame
SPORTx_CHNL	SPORTx current channel registers on page 30-71	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.

SPORTx_TCR1 and SPORTx_TCR2 Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORTx_TCR1 and SPORTx_TCR2, shown in Figure 30-25 and Figure 30-26.

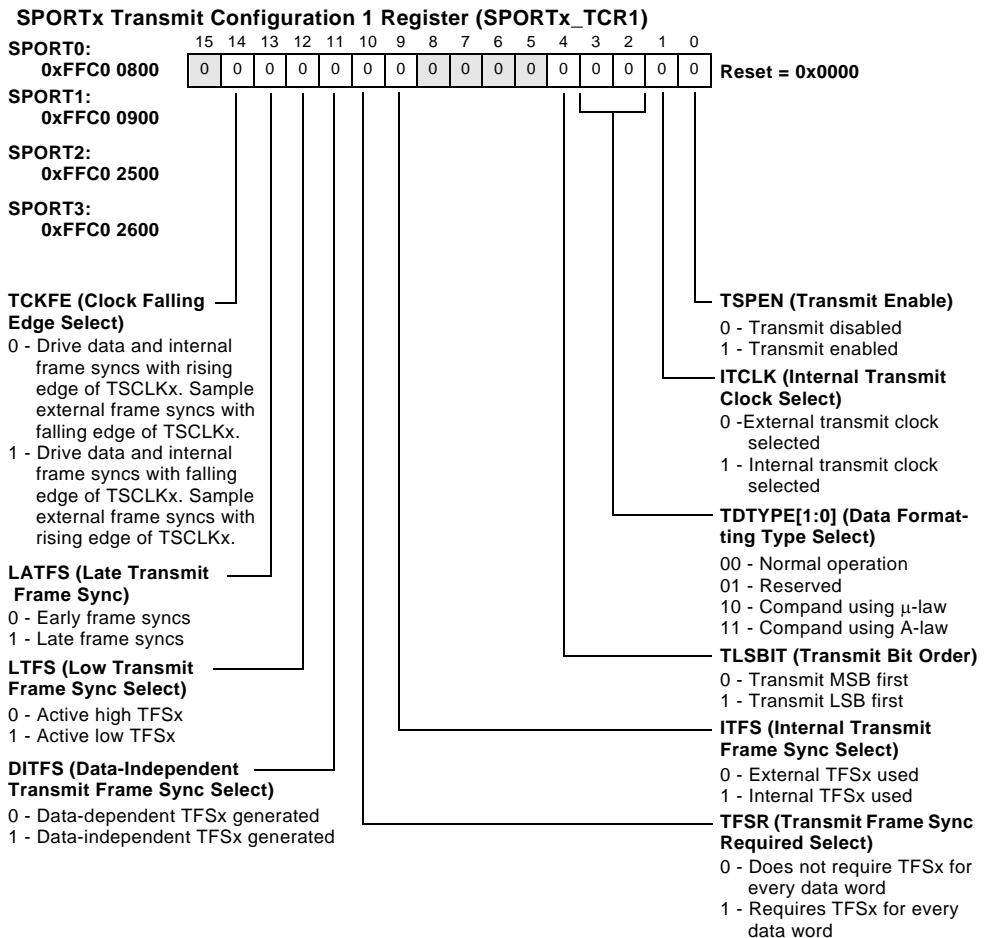


Figure 30-25. SPORTx Transmit Configuration 1 Register

SPORT Registers

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, 0xFFFF) ; /* SPORT disabled, SPORTx_TCR1
                                still equal to 0x0000 */
```

SPORTx Transmit Configuration 2 Register (*SPORTx_TCR2*)

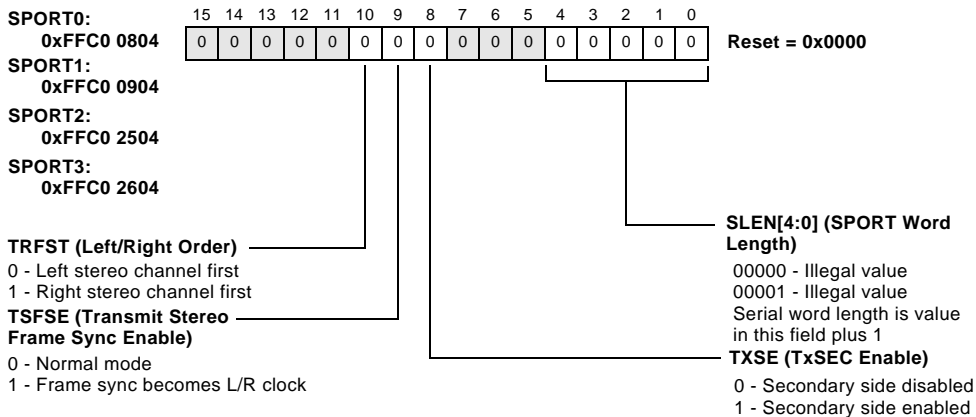


Figure 30-26. SPORTx Transmit Configuration 2 Register

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.

SPORT Registers


- **Serial word length select.** (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

$$\text{Serial Word Length} = \text{SLEN} + 1;$$

The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer 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 SPORT_x_TFSDIV and SPORT_x_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal transmit frame sync select.** (ITFS). This bit selects whether the SPORT uses an internal TFS_x (if set) or an external TFS_x (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).

SPORT Registers

- **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 30-27](#) and [Figure 30-28](#).

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, 0xFFFF0) ; /* SPORT disabled, SPORTx_RCR1
                                still equal to 0x0000 */
```

SPORTx Receive Configuration 1 Register (SPORTx_RCR1)

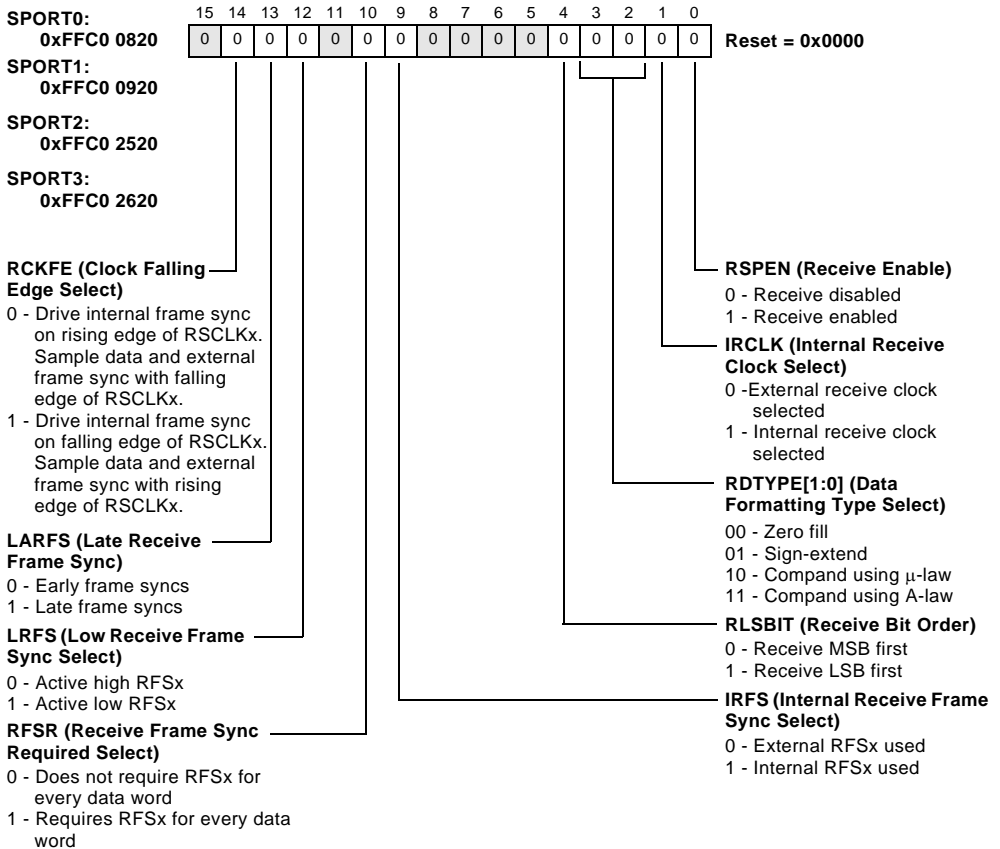


Figure 30-27. SPORTx Receive Configuration 1 Register

SPORT Registers

SPORTx Receive Configuration 2 Register (SPORTx_RCR2)

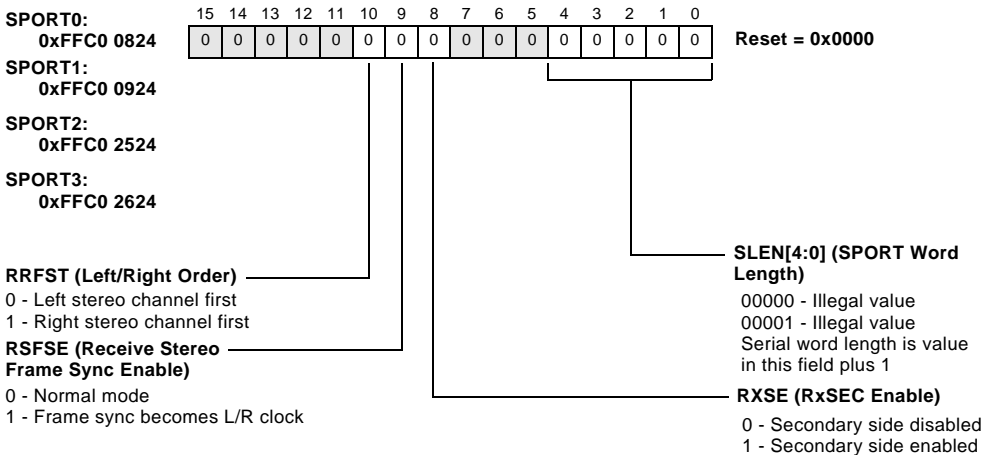


Figure 30-28. SPORTx Receive Configuration 2 Register

Additional information for the `SPORTx_RCR1` and `SPORTx_RCR2` receive configuration register bits:

- **Receive enable.** (`RSPEN`). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the `RSPEN` bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

Setting `RSPEN` enables the SPORTx receiver, which can generate a SPORTx RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting `RSPEN`. Setting `RSPEN` also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting `RSPEN`.

Clearing `RSPEN` causes the SPORT to stop receiving data; it also

shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing `RSPEN` whenever the SPORT is not in use.



All SPORT control registers should be programmed before `RSPEN` is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write `SPORTx_RCR1` with all of the necessary bits, including `RSPEN`.

- **Internal receive clock select.** (`IRCLK`). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The `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).

SPORT Registers

- **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.

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 30-29](#). The SPORTx_TX register is shown in [Figure 30-30](#).

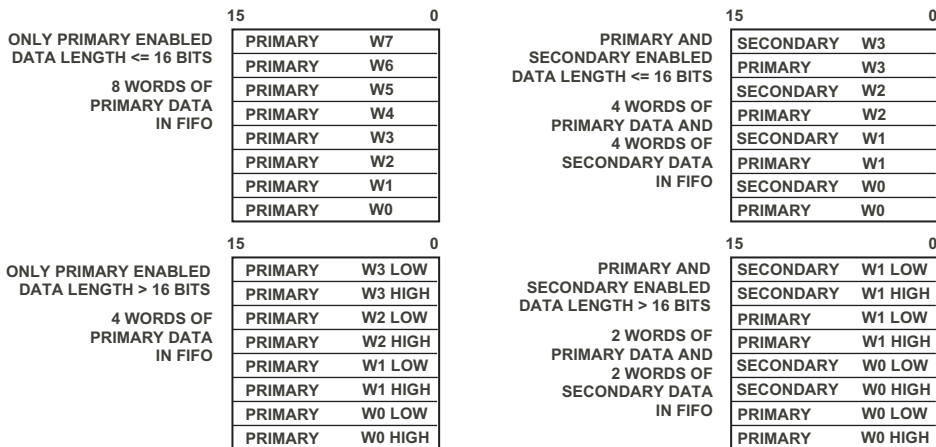


Figure 30-29. SPORT Transmit FIFO Data Ordering

SPORT Registers

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that PAB/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/PAB writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on `TXSE` and `SLEN`, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the `DTPRI` and `DTSEC` pins.

The SPORT TX interrupt is asserted when `TSPEN = 1` and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled. For DMA operation, see the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference*.

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 `TXEN = 0`).

If software causes the core processor to attempt a write to a full TX FIFO with a `SPORTx_TX` write, the new data is lost and no overwrites occur to data in the FIFO. The `TOVF` status bit is set and a SPORT error interrupt is asserted. The `TOVF` bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the `SPORTx_TX` register without causing this type of error, read the register's status first. The `TXF` bit in the SPORT status register is 0 if space is available for another word in the FIFO.

The `TXF` and `TOVF` status bits in the `SPORTx` status register are updated upon writes from the core processor, even when the SPORT is disabled.

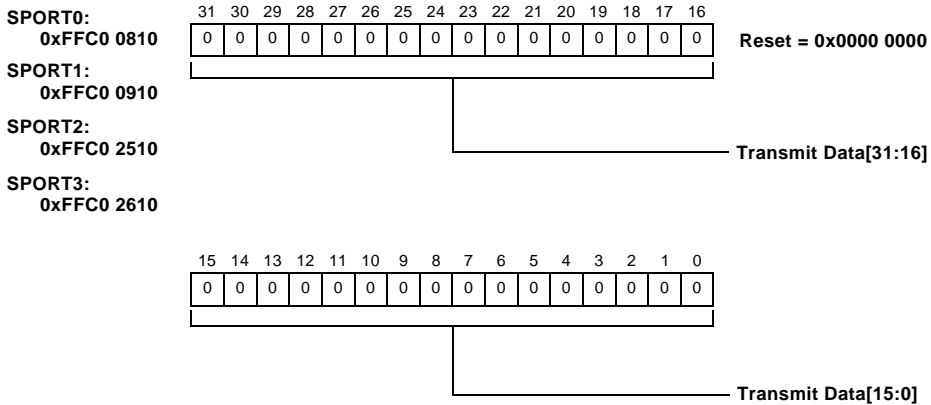
SPORTx Transmit Data Register (SPORTx_TX)

Figure 30-30. SPORTx Transmit Data Register

SPORTx_RX Register

The SPORTx receive data register (SPORTx_RX) is a read-only register. Writes produce a PAB error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length ≤ 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using PAB/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO are shown in [Figure 30-31](#). The SPORTx_RX register is shown in [Figure 30-32](#).

SPORT Registers

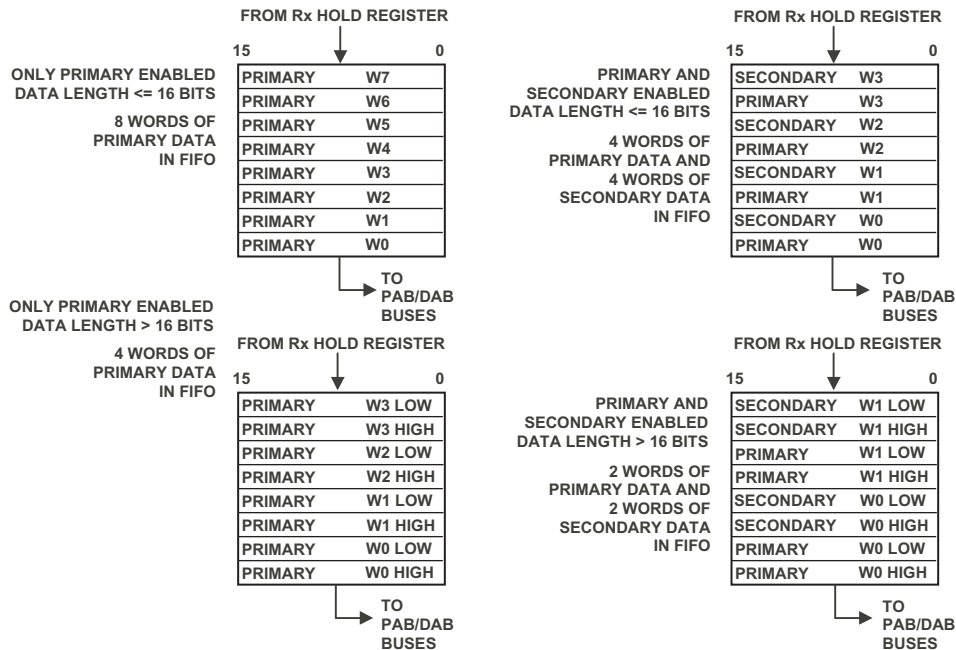


Figure 30-31. SPORT Receive FIFO Data Ordering

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.

SPORTx Receive Data Register (SPORTx_RX)

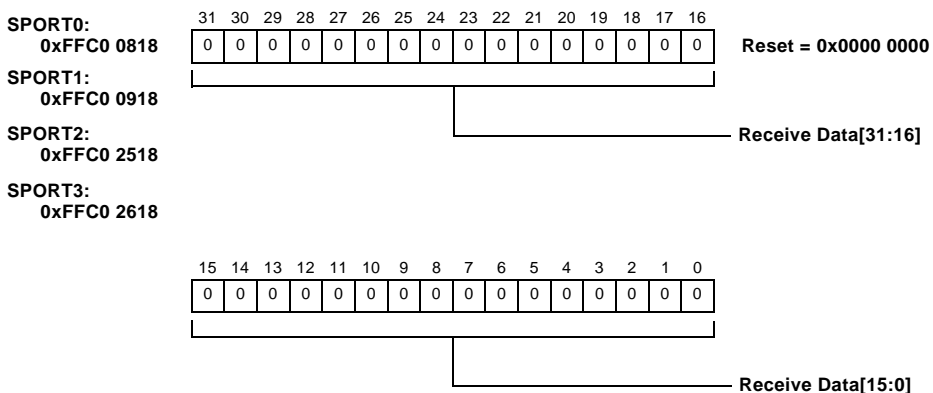


Figure 30-32. SPORTx Receive Data Register

SPORTx_STAT Register

The SPORT status register (`SPORTx_STAT`) is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. This register is shown in [Figure 30-33](#).

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 `TXEN = 0`).

For continuous transmission (`TFSR = 0`), `TUVF` is set at the end of a transmitted word if no new word is available in the TX hold register.

The `TOVF` bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing `TXEN = 0`. Both `TXF` and `TOVF` are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (`ROVF`) is set in the SPORT status register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing `RXEN = 0`).

The `RUVF` bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing `RXEN = 0`. The `RUVF` bit is updated even when the SPORT is disabled.

SPORTx Status Register (SPORTx_STAT)

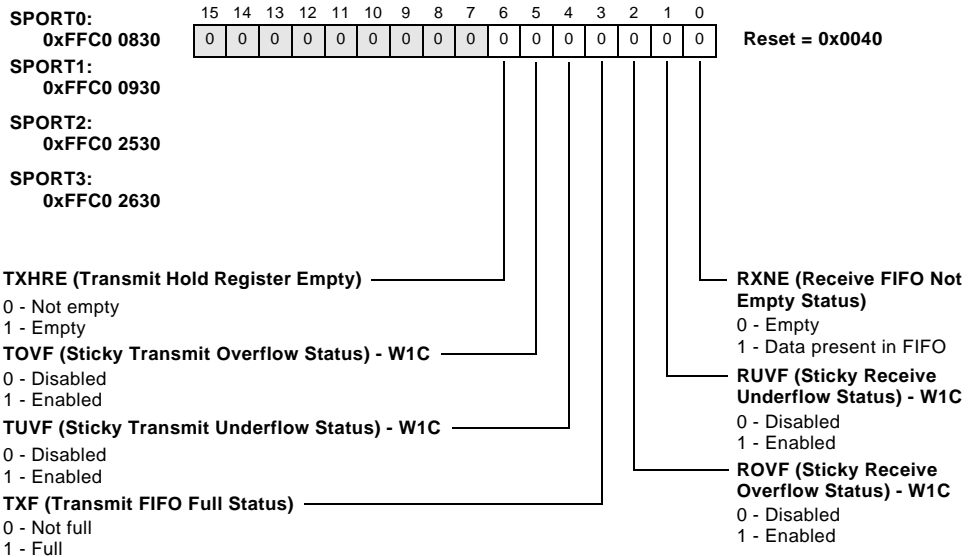


Figure 30-33. SPORTx Status Register

SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORTx transmit serial clock divider register, SPORTx_TCLKDIV, shown in [Figure 30-34](#), and the SPORTx receive serial clock divider register, SPORTx_RCLKDIV, shown in [Figure 30-35](#)).

SPORTx Transmit Serial Clock Divider Register (SPORTx_TCLKDIV)

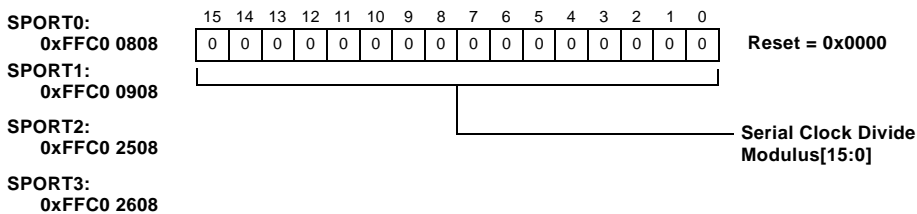


Figure 30-34. SPORTx Transmit Serial Clock Divider Register

SPORTx Receive Serial Clock Divider Register (SPORTx_RCLKDIV)

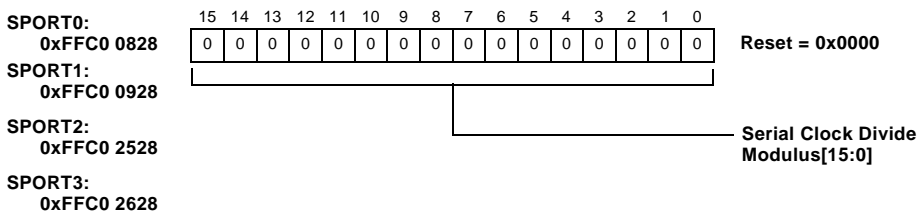


Figure 30-35. SPORTx Receive Serial Clock Divider Register

SPORTx_TFSDIV and SPORTx_RFSDIV Register

The 16-bit SPORTx transmit frame sync divider register (SPORTx_TFSDIV) and the SPORTx receive frame sync divider register (SPORTx_RFSDIV) specify how many transmit or receive clock cycles are counted before generating a 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 30-36 and Figure 30-37.

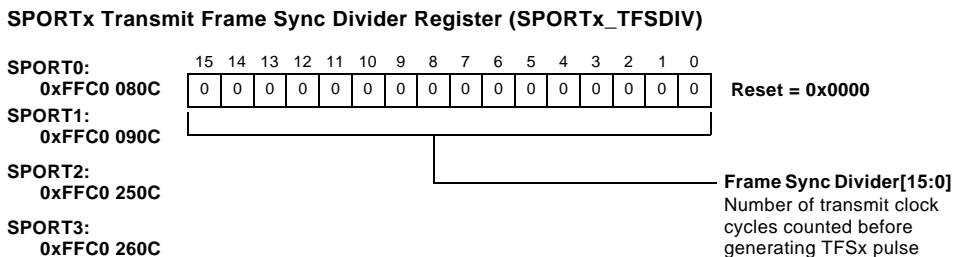


Figure 30-36. SPORTx Transmit Frame Sync Divider Register

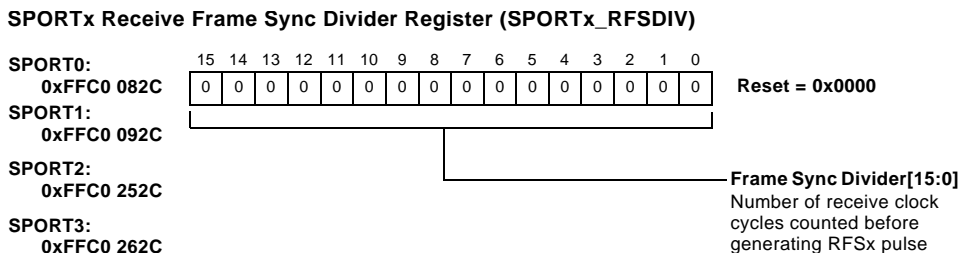


Figure 30-37. SPORTx Receive Frame Sync Divider Register

SPORTx_MCMCn Registers

There are two SPORTx multichannel configuration registers (SPORTx_MCMCn) for each SPORT, shown in [Figure 30-38](#) and [Figure 30-39](#). The SPORTx_MCMCn registers are used to configure the multichannel operation of the SPORT.

SPORTx Multichannel Configuration Register 1 (SPORTx_MCMC1)

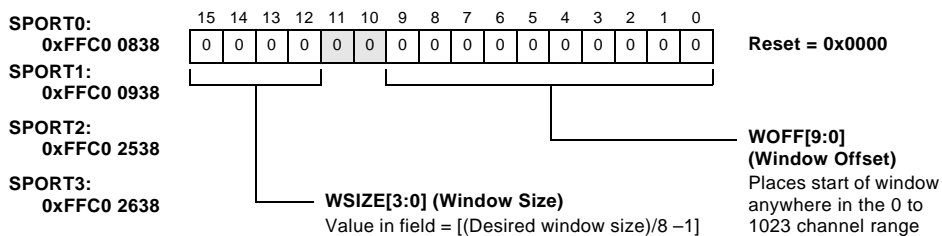


Figure 30-38. SPORTx Multichannel Configuration Register 1

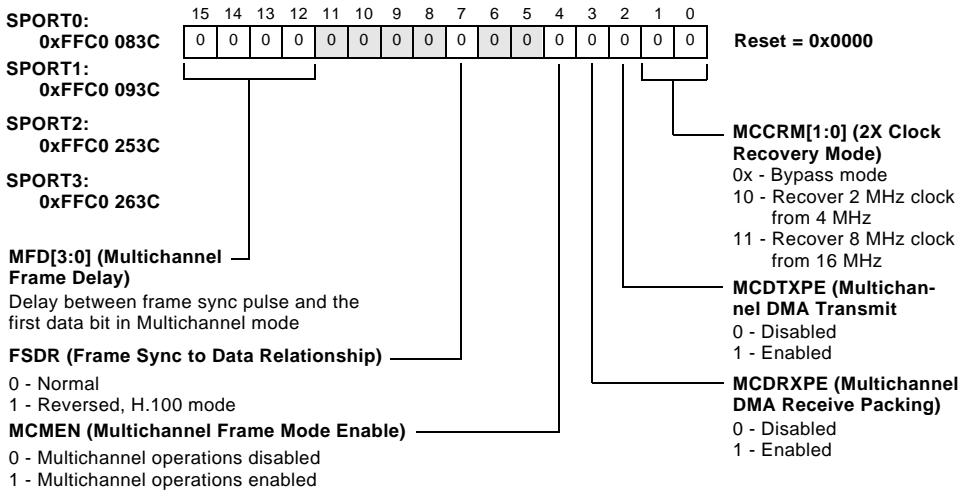
SPORTx Multichannel Configuration Register 2 (SPORTx_MCMC2)

Figure 30-39. SPORTx Multichannel Configuration Register 2

SPORTx_CHNL Register

The 10-bit **CHNL** field in the SPORTx current channel register (**SPORTx_CHNL**) indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The **CHNL[9:0]** field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between **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.

SPORT Registers

The `SPORTx_CHNL` register is shown on [Figure 30-40](#).

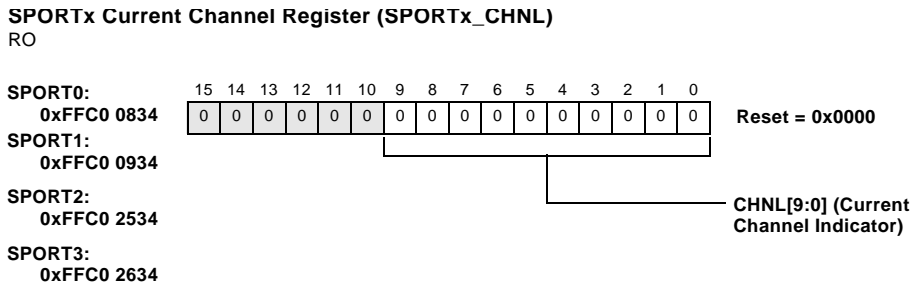


Figure 30-40. SPORTx Current Channel Register

SPORTx_MRCSn Registers

The multichannel selection registers are used to enable and disable individual channels. The SPORTx multichannel receive select registers (`SPORTx_MRCSn`, shown in [Figure 30-41](#)) 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 30-6](#) 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 30-6](#).

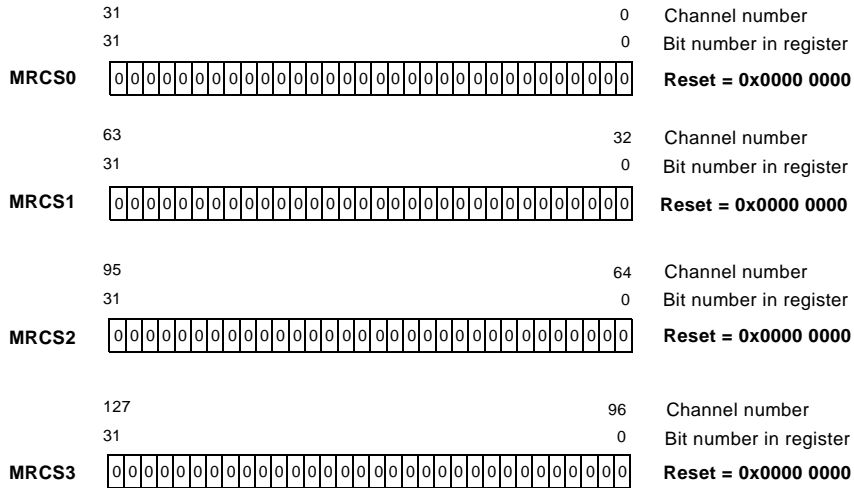


Figure 30-41. SPORTx Multichannel Receive Select Registers

SPORT Registers

Table 30-6. SPORT_x 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

SPORT_x_MTCS_n Registers

The multichannel selection registers are used to enable and disable individual channels. The four SPORT_x multichannel transmit select registers (SPORT_x_MTCS_n, [Figure 30-42](#)) 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 SPORT_x_MTCS_n 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 SPORT_x_MTCS_n 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 30-7](#).

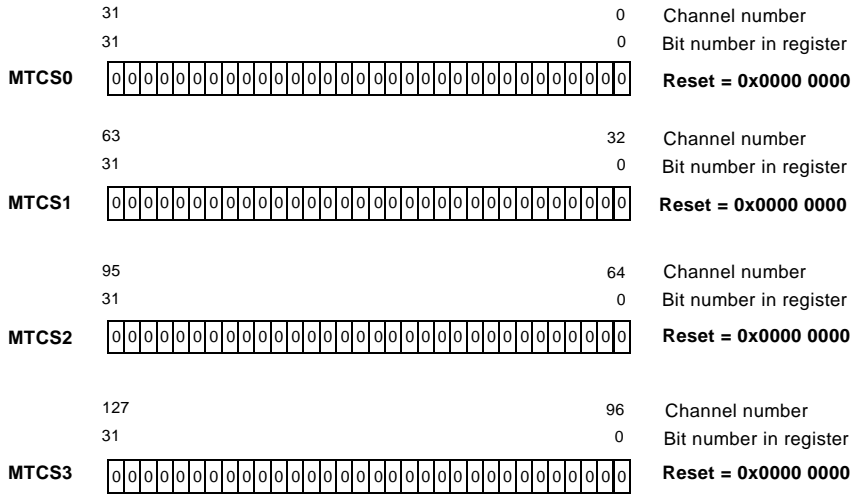


Figure 30-42. SPORTx Multichannel Transmit Select Registers

Programming Examples

Table 30-7. SPORT_x 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 30-1](#) through [Listing 30-4](#) on [page 30-82](#) 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 (SPORT_x_RCR_x and SPORT_x_TCR_x) 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 PORT_x_FER and PORT_x_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 30-1. 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) */

```

Programming Examples

```
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 a detailed explanation of the bits, see the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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 30-2. 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;

```

Programming Examples

```
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 30-3. 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 30-4. 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;
```

31 UART PORT CONTROLLERS

This chapter describes the universal asynchronous receiver/transmitter (UART) modules and includes the following sections:

- [“Overview” on page 31-1](#)
- [“Interface Overview” on page 31-3](#)
- [“Description of Operation” on page 31-6](#)
- [“Programming Model” on page 31-23](#)
- [“UART Registers” on page 31-28](#)
- [“Programming Examples” on page 31-51](#)

Overview

The ADSP-BF54x 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 31-1 shows a simplified block diagram of one UARTx module and how it interconnects to the Blackfin architecture and to the outside world.

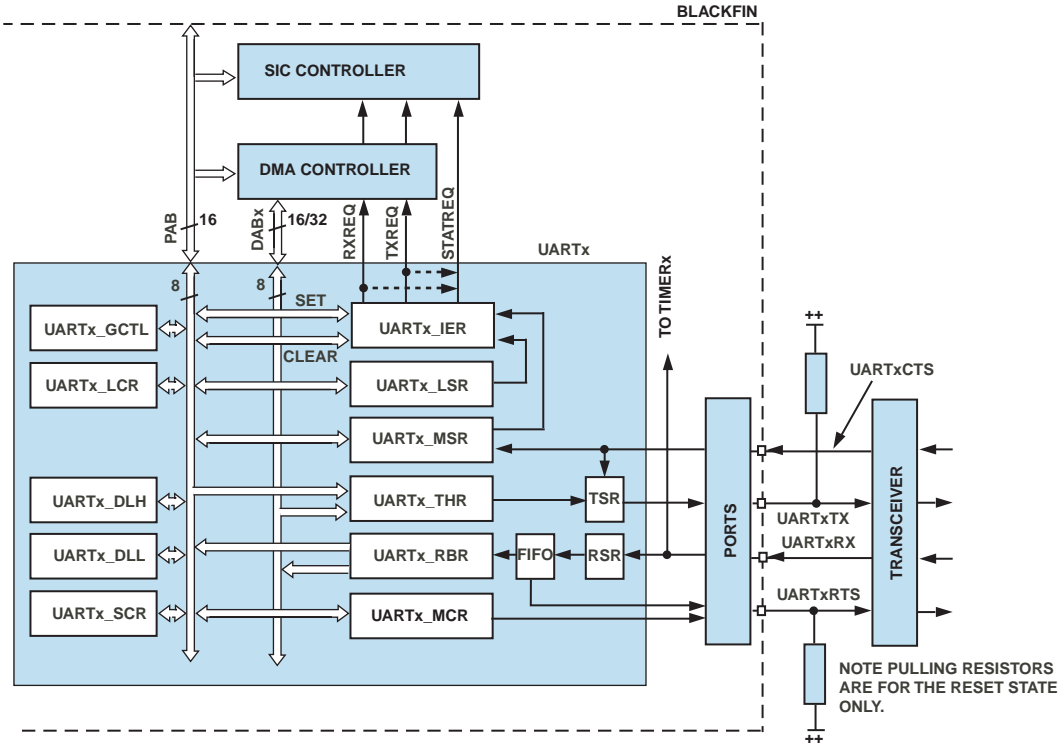


Figure 31-1. UART Block Diagram

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 31-1](#) shows where the signal can be found and how they are enabled in the port control.

Table 31-1. 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)

Table 31-1. UART Signals (Cont'd)

Signal	Pin	Port Control	Autobaud Timer
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 (TAC13)
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 on DMA, see the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.


All UART registers are 8 bits wide. They connect to the PAB bus. The UART_x_RBR and UART_x_THR registers also connect to one of the DAB_x buses. 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 DMA_x_PERIPHERAL_MAP register of the desired DMA channels.

Description of Operation

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 31-1](#) 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 31-21](#).

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 31-2](#) shows a typical physical bitstream measured on one of the TX pins.

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.

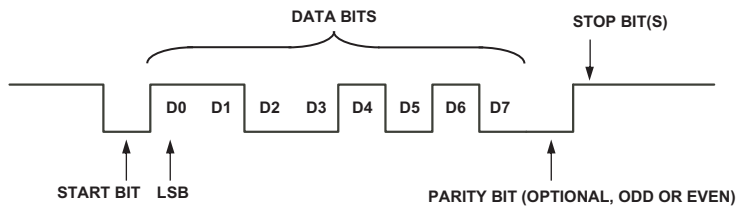


Figure 31-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

IrDA support is enabled by setting the `IREN` bit in the `UARTx_GCTL` register. The IrDA application requires external transceivers.

UART Transmit Operation

Receive and transmit paths operate completely independently except that the bit rate and the frame format are identical for both transfer directions.

Description of Operation

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:

$$\text{BIT RATE} = \frac{\text{SCLK}}{16^{(1-\text{EDBO})} \times \text{Divisor}}$$

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 `EDB0` 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:

$$\text{BIT RATE} = \frac{\text{SCLK}}{16^{(1-\text{EDB0})} \times \text{Divisor}}$$

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 31-13](#). 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

Description of Operation

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 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 31-3](#).

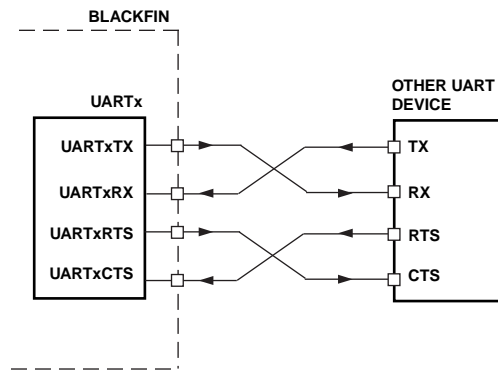


Figure 31-3. UART Hardware Flow

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 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 `UARTxCTS` 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.

Description of Operation

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.

i 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 31-4. The final IrDA pulse is fed to the off-chip infrared driver.

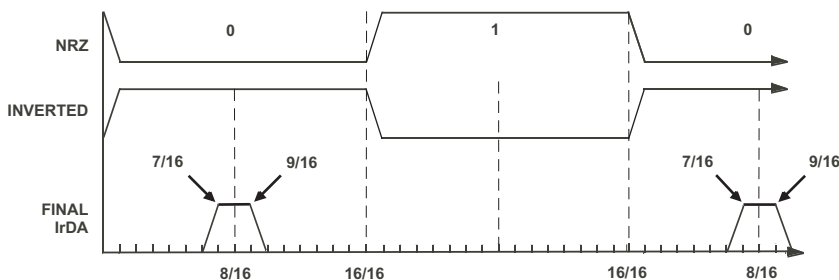


Figure 31-4. IrDA Transmit Pulse

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 31-2 on page 31-20](#), 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.

Description of Operation

The polarity of receive data is selectable, using the `IRPOL` bit. Figure 31-5 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.

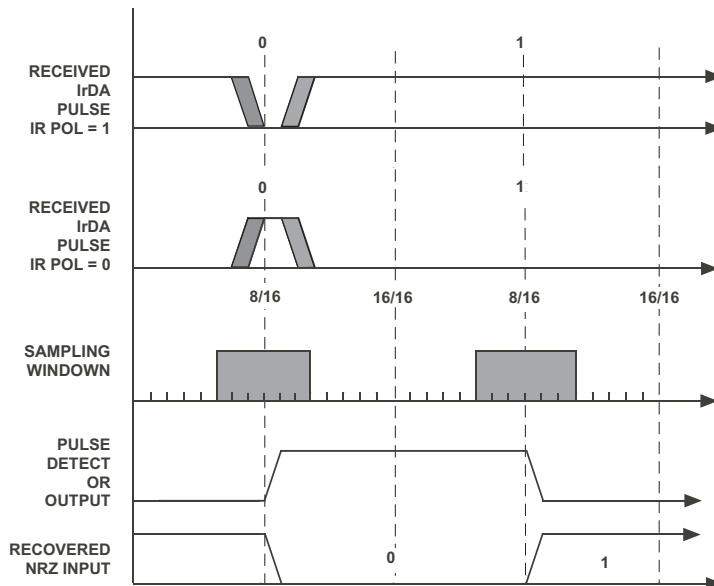



Figure 31-5. IrDA Receiver Pulse Detection

i In the IrDA mode the `EDB0` bit is ignored. The sample frequency is always exactly 16 times the bit rate.

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 31-1 on page 31-3](#), 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. Refer to the description of the peripheral map registers in the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

 On ADSP-BF54x 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 31-1 on page 31-3](#)). 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.

Description of Operation

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 31-25](#).

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

“[UARTx_LSR Registers](#)” on page 31-36 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.

Description of Operation

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 EDB0 bit in the UARTx_GCTL register is set, so that the following applies:


$$\text{BIT RATE} = \frac{\text{SCLK}}{16^{(1-\text{EDB0})} \times \text{Divisor}}$$

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

Table 31-2 provides example divide factors required to support most standard baud rates.

Table 31-2. 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

-  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 ($EDB0=1$) 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 $EDB0=1$ mode only when bit rate accuracy is not acceptable in $EDB0=0$ mode.

The $EDB0=1$ mode is not intended to increase operation speed beyond the electrical limitations of the asynchronous UART transfer protocol.

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 WDT_CAP mode these timers can be used to automatically detect the bit rate applied to the UART_xRX pin by an external device. For more information, see the “General-Purpose Timers” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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

Description of Operation

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:

$$\text{DIVISOR} = \frac{\text{TIMERx_WIDTH}}{16^{(1-\text{EDB0})} \times \text{Number of captured UART bits}}$$

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 31-6](#).

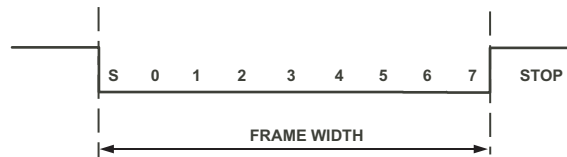


Figure 31-6. Autobaud Detection Character 0x00

Because the example frame in [Figure 31-6](#) encloses 8 data bits and 1 start bit, apply the following formula:

$$\text{DIVISOR} = \frac{\text{TIMERx_WIDTH}}{16^{(1-\text{EDB0})} \times 9}$$

Real `UARTxRX` 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 31-7](#), 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 `EDB0 = 0`
- Divisor = `TIMERx_PERIOD >> 3` if `EDB0 = 1`

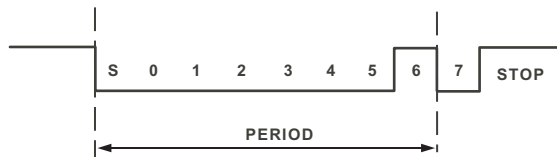


Figure 31-7. Autobaud Detection Character 0x40

An example is provided in [Listing 31-2 on page 31-53](#).

Programming Model

The following sections describe a programming model for the UARTs.

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into `UARTx_THR`. Received data can be read from `UARTx_RBR`. The processor must write and read a limited number of characters at a time.

Programming Model

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, `UARTxRX`, 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 31-15 on page 31-45](#).

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 the “Direct Memory Access” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

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

Programming Model

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 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 31-3](#) provides an overview of the UART registers.

Unlike on ADSP-BF53x processors, register addresses are not shared on ADSP-BF54x processors. Each register has its own MMR address. Consequently, the `DLAB` bit is not present on ADSP-BF54x 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 31-3. ADSP-BF54x vs. ADSP-BF53x UART Register

Name	ADSP-BF54x Address Offset	ADSP-BF53x Address Offset	Register Name
<code>UARTx_DLL</code>	0x00	0x00, <code>DLAB=1</code>	UART divisor latch low byte registers on page 31-48
<code>UARTx_DLH</code>	0x04	0x00, <code>DLAB=1</code>	UART divisor latch high byte registers on page 31-48

Table 31-3. ADSP-BF54x vs. ADSP-BF53x UART Register (Cont'd)

Name	ADSP-BF54x Address Offset	ADSP-BF53x Address Offset	Register Name
UARTx_GCTL	0x08	0x24	UART global control register on page 31-50
UARTx_LCR	0x0C	0x0C	UART line control registers on page 31-30
UARTx_MCR	0x10	0x10	UART modem control registers on page 31-33
UARTx_LSR	0x14	0x14	UART line status registers on page 31-36
UARTx_MSR	0x18	N/A	UART modem status registers on page 31-39
UARTx_SCR	0x1C	0x1C	UART scratch registers on page 31-49
UARTx_IER_SET	0x20	N/A	UART interrupt enable set registers on page 31-43
UARTx_IER_CLEAR	0x24	N/A	UART interrupt enable clear registers on page 31-43
UARTx_IER	N/A	0x04, DLAB=0	Interrupt Enable R/W register on page 31-30
UARTx_THR	0x28	0x00, DLAB=0	UART transmit hold registers on page 31-41
UARTx_RBR	0x2C	0x00, DLAB=0	UART receive buffer registers on page 31-42
UARTx_IIR	N/A	0x08	Interrupt Enable register on page 31-30

UART Registers

UARTx_LCR Registers

The line control (UARTx_LCR) registers, shown in [Figure 31-8](#), control the format of received and transmitted character frames.

UART Line Control Registers (UARTx_LCR)

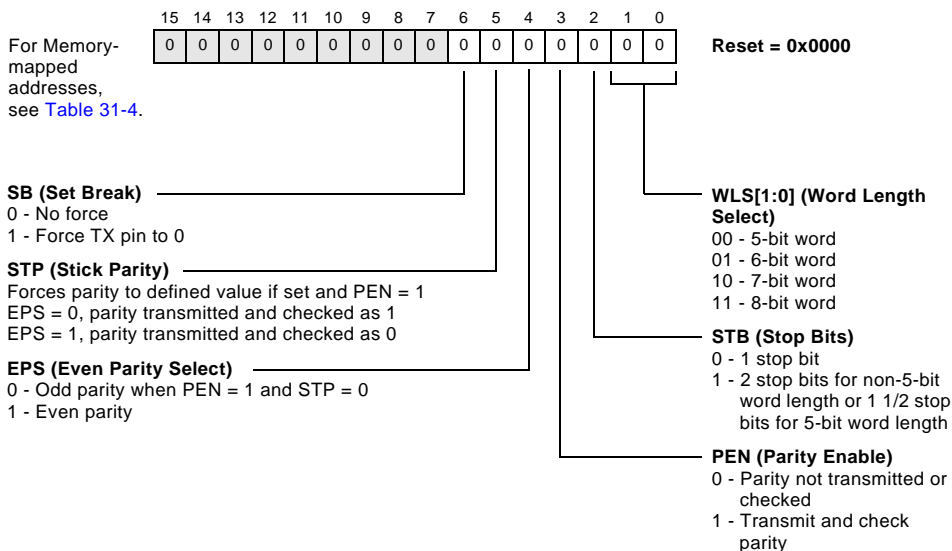


Figure 31-8. UART Line Control Registers

Table 31-4. 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 31-5](#) summarizes polarity behavior assuming 8-bit data words (`WLS=3`).

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.

UART Registers

Table 31-5. 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
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

UARTx_MCR Registers

The modem control (UARTx_MCR) registers control the UART port, as shown in Figure 31-9. Partial modem functionality is supported to allow for hardware flow control and loopback mode.

UART Modem Control Registers (UARTx_MCR)

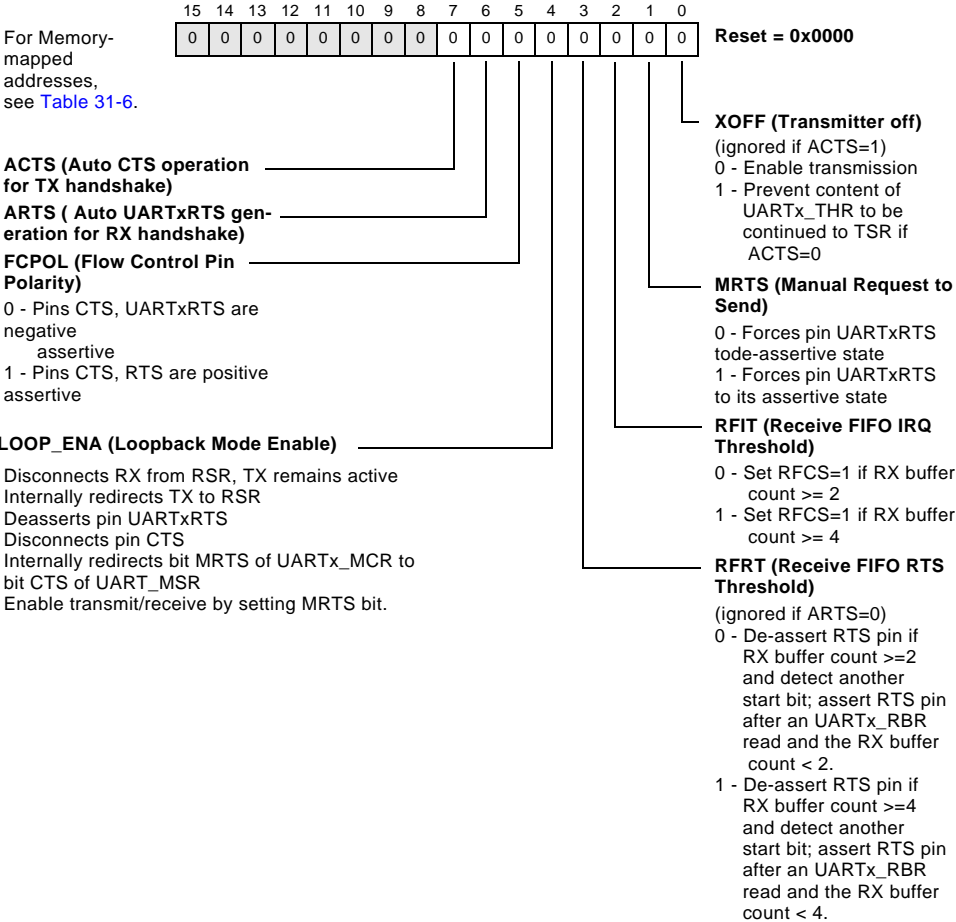


Figure 31-9. UART Modem Control Registers

UART Registers

Table 31-6. 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 `ACTS=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.

UART Registers

UARTx_LSR Registers

The line status (UARTx_LSR) registers contain UART status information as shown in Figure 31-10. Unlike the industrial standard, the ADSP-BF54x 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.

UART Line Status Registers (UARTx_LSR)

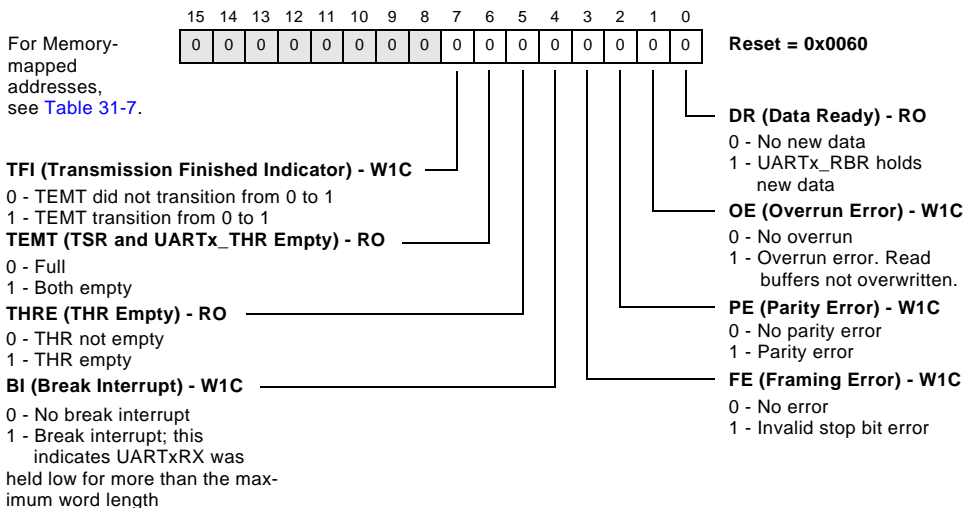


Figure 31-10. UART Line Status Registers

Table 31-7. 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.

UART Registers

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.

UARTx_MSR Registers

The modem status (UARTx_MSR) registers, shown in Figure 31-12, contains current states of the UART’s external UARTxCTS pin and current status of the UART's internal receive buffers.

UART Modem Status Registers (UARTx_MSR)

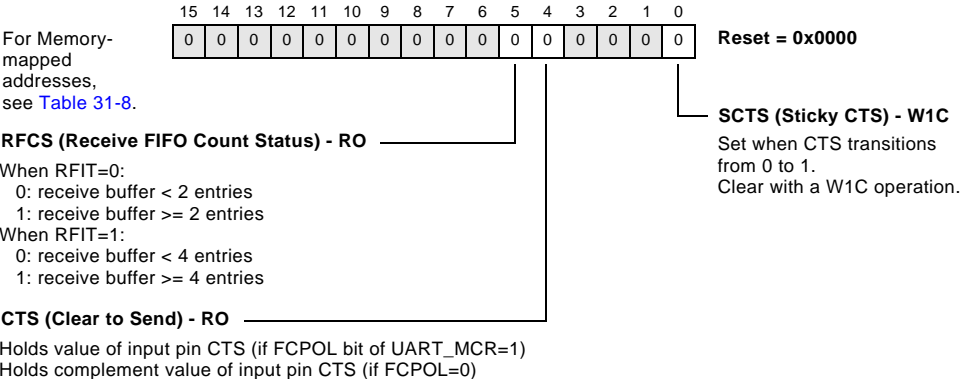


Figure 31-11. UART Modem Status Registers

Table 31-8. 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


UART Registers

= 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.

UARTx_THR Registers

The write-only transmit hold (UARTx_THR) registers, shown in [Figure 31-12](#), 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.

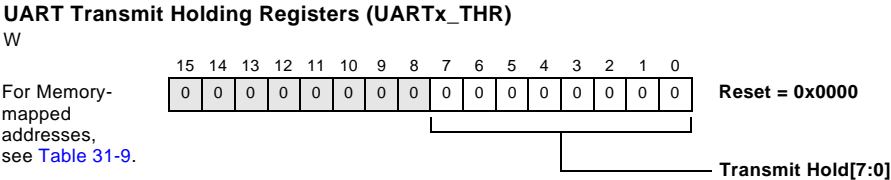


Figure 31-12. UART Transmit Holding Registers

Table 31-9. 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

UART Registers

UARTx_RBR Registers

The read-only `UARTx_RBR` registers, shown in [Figure 31-13](#), 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.

UART Receive Buffer Registers (UARTx_RBR)

RO

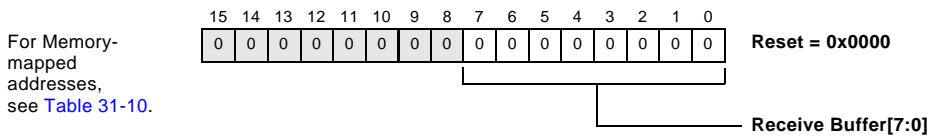


Figure 31-13. UART Receive Buffer Registers

Table 31-10. 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

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 31-14](#) and [Figure 31-15](#), 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.

UART Registers

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)

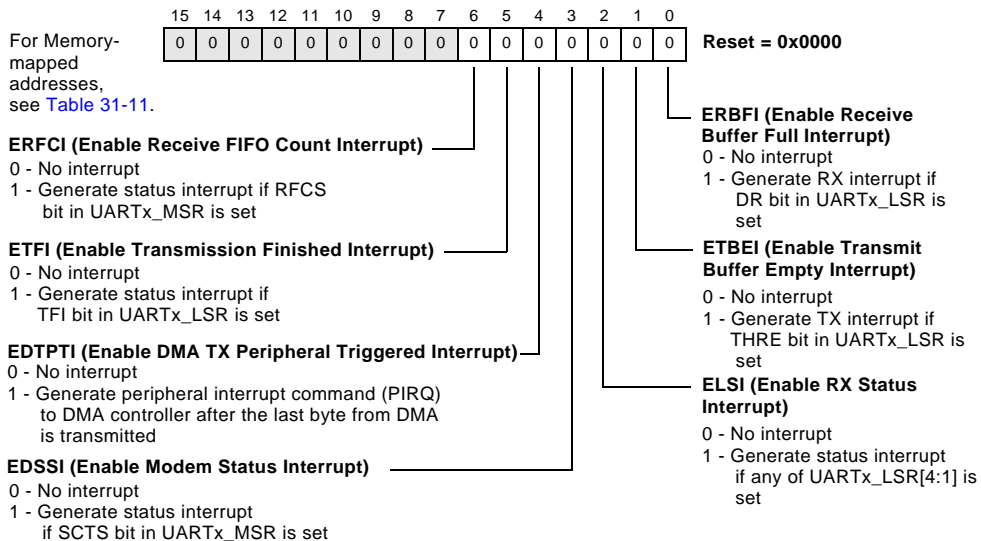


Figure 31-14. UART Interrupt Enable Set Registers

Table 31-11. 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

UART Interrupt Enable Clear Registers (UARTx_IER_CLEAR)

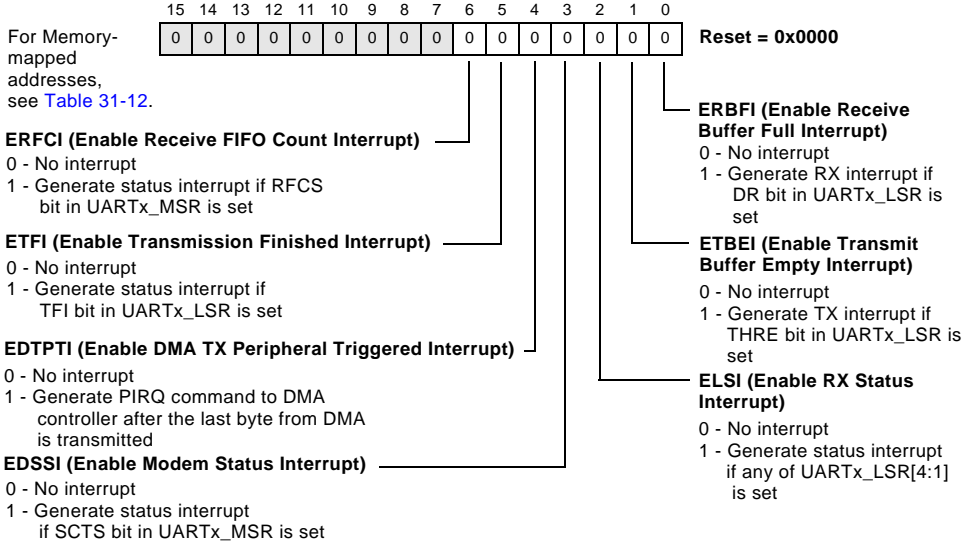


Figure 31-15. UART Interrupt Enable Clear Registers

Table 31-12. 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,

UART Registers

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.



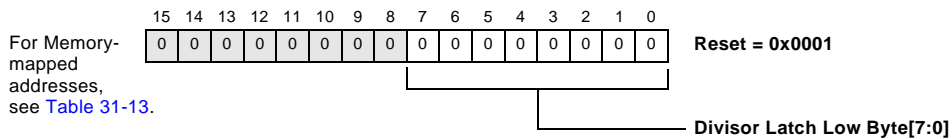
The `UARTx_IIR` registers are not present on this implementation. Signalling interrupt sources can be identified by interrogating `UARTx_LSR` and `UARTx_MSR` status registers.

UART Registers

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 31-16](#).

UART Divisor Latch Low Byte Registers (UARTx_DLL)



UART Divisor Latch High Byte Registers (UARTx_DLH)

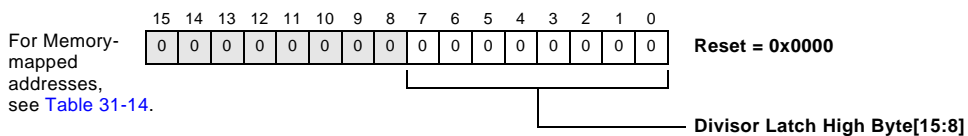


Figure 31-16. UART Divisor Latch Registers

Table 31-13. 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 31-14. 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

i 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 `EDB0` bit in the `UARTx_GCTL` register. Refer to “Bit Rate Generation” on page 31-19.

UARTx_SCR Registers

The contents of the 8-bit scratch (`UARTx_SCR`) registers, shown in Figure 31-17, are reset to 0x00. They are used for general-purpose data storage and do not control the UART hardware in any way.

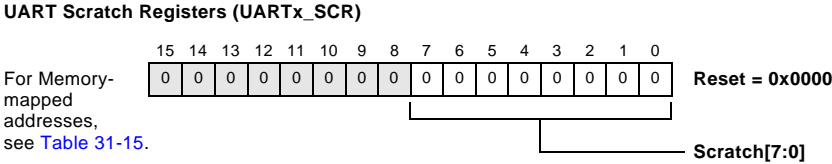


Figure 31-17. UART Scratch Registers

UART Registers

Table 31-15. 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

UARTx_GCTL Registers

The global control (UARTx_GCTL) registers, shown in Figure 31-18, contain the enable bit for internal UART clocks and for the IrDA mode of operation of the UARTs.

UART Global Control Registers (UARTx_GCTL)

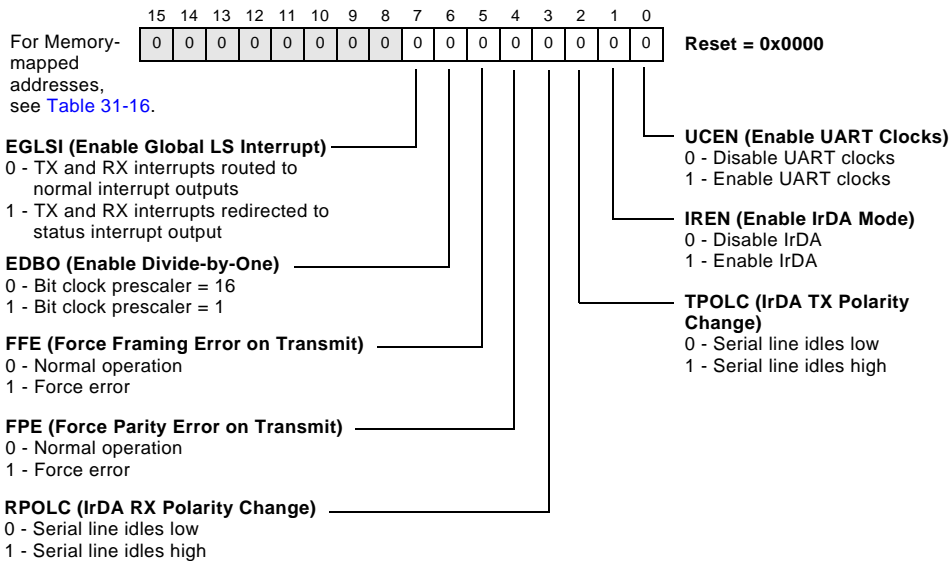


Figure 31-18. UART Global Control Registers

Table 31-16. 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 `EDB0` 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 31-19](#). 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.

Programming Examples

The subroutine in [Listing 31-1](#) shows a typical UART initialization sequence.

Listing 31-1. 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 31-2](#) performs autobaud detection similarly to UART boot.

Listing 31-2. 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 | TIMILO (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 */

```

Programming Examples

```
r7 = TIN_SEL | IRQ_ENA | PERIOD_CNT | WDT_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 31-3](#) performs autobaud detection using UART0 and TIMER0.

Listing 31-3. 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(TIMERO_CONFIG); /* select TIMER 0 */
    p1.h = hi(TIMERO_CONFIG);
    call uart_autobaud;
    r0 >>= 7;          /* divide PERIOD value by (16 x 8) */
    call uart_init;
    ...

```

The subroutine in [Listing 31-4](#) transmits a character by polling operation.

Listing 31-4. 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
*/

```

Programming Examples

```
    r7 = [sp++];  
    rts;  
uart_putc.end:
```

Use the routine shown in [Listing 31-5](#) to transmit a C-style string that is terminated by a null character.

Listing 31-5. 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 31-6](#).

Listing 31-6. 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++];

```

Programming Examples

```
        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 31-7 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 31-7. 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);

```

Programming Examples

```
    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:
```


32 USB OTG CONTROLLER

This chapter describes the USB OTG interface. This peripheral is a 6-pin interface for the USB OTG controller.

This chapter includes the following sections:

- [“Overview” on page 32-2](#)
- [“Interface Overview” on page 32-3](#)
- [“Description of Operation” on page 32-12](#)
- [“Functional Description” on page 32-44](#)
- [“Programming Model” on page 32-46](#)
- [“USB OTG Registers” on page 32-64](#)
- [“Programming Examples” on page 32-141](#)
- [“References” on page 32-141](#)
- [“Glossary of USB Terms ” on page 32-141](#)

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 to the USB 2.0 Specification. In host mode, the USB module supports transfers at high-speed (480Mbps), full-speed (12Mbps), and low-speed (1.5Mbps)

Overview

rates. peripheral mode supports the high- and full-speed transfer rates. Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The 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 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
- 1 bidirectional control endpoint
- Seven transmit and seven receive unidirectional endpoints
- 7.232 KBytes of FIFOs for packet buffering
- 8 DMA master channels

- 3 top-level maskable general purpose interrupts
- 1 asynchronous wakeup interrupt
- 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
- High-bandwidth isochronous and interrupt endpoint support
- 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.

Interface Overview

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 and in the other direction USB OUT packets are received into the Rx FIFOs (having being 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 and 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 32-1](#) 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 8 DMA request lines. Three top-level maskable interrupts are provided each of which can be source 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 32-8](#).

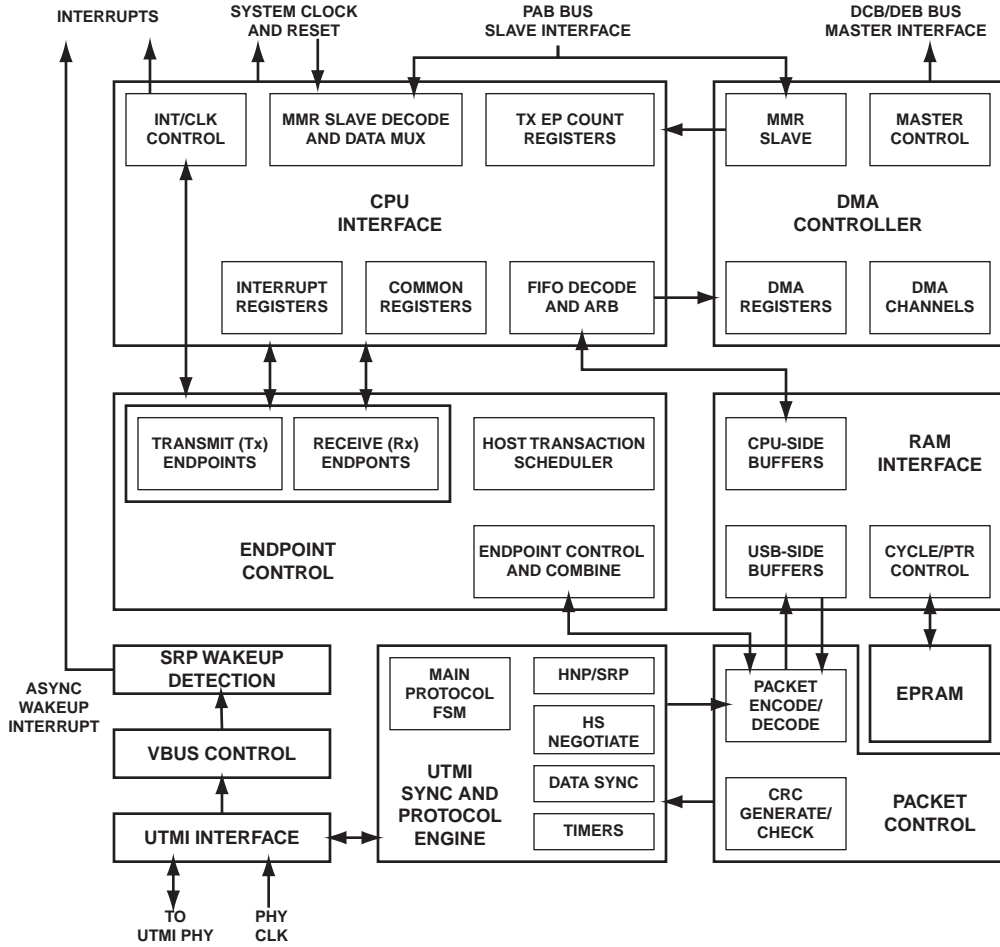


Figure 32-1. USB OTG Controller Block Diagram

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 32-bits wide.


Interface Overview

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. 7.232kBytes of SRAM are available.

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 32-46](#).

The USB controller requires a system clock frequency of >30MHz 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 asynchronous wakeup circuit is used to detect when another 'B' device is asserting its D+ pullup to initiate SRP when all other clocks are off. This circuit requires a slow clock (for example, 32kHz).

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.

Before any endpoint register writes can be committed on endpoint zero (control transfers) take place, the GLOBAL_ENA bit of the USB_GLOBAL_CTL register must be set in order to enable the system clock to 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 default 'A' device powering the bus), to discharge VBUS (to speed 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 32-124 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 endpoint 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 32-1](#) lists the endpoint FIFO configuration, with an indication of the transfer types possible for that particular buffer size.

Table 32-1. 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

Interface Overview

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); and
- 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 is route-able 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 are not route-able 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 32-2](#) 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.

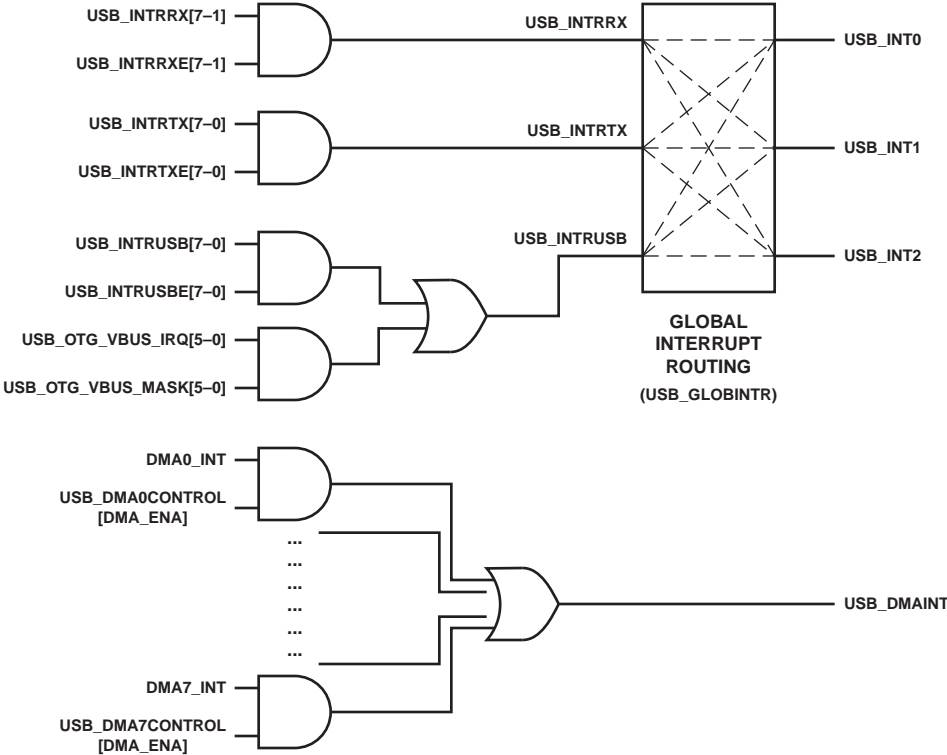


Figure 32-2. USB Interrupt Sources and Routing

Interrupts can be generated from control endpoint zero from 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.

Interface Overview

Interrupts can be generated from transmit endpoints (USB_INTRTX) from 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) from 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) from 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 from the USB controller:

- Drive VBUS >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 2.5µs 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

Description of Operation

- 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 any need to switch the cable connecting the two devices. The conditions under which the USB controller may switch between peripheral and host mode are detailed [“Host Negotiation/Configuration” on page 32-49](#).

Peripheral Mode Operation

Operations for the USB OTG interface when in 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 such as the size of hardware FIFO available and system latencies as well as the USB transfer type and class being used. As far as USB is concerned, 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 32-1 on page 32-7](#)). Because the USB controller uses a 16-bit interface, it is recommended that the value chosen for *MaxPktSize* is 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.

Description of Operation

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 32-64 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 32-1](#) on page 32-7).

 Note that 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 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` 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 clear then there are no packets in the FIFO and two more packets can be loaded.

Description of Operation

High Bandwidth Isochronous IN Endpoints

In high-speed mode, isochronous TX endpoints can transmit up to three USB packets in any micro-frame, with a payload of up to 1024/3 bytes in each packet, corresponding to a data transfer rate of up to 1024 bytes per micro-frame. [Figure 32-3](#) provides an overview of high-bandwidth IN endpoints in USB.

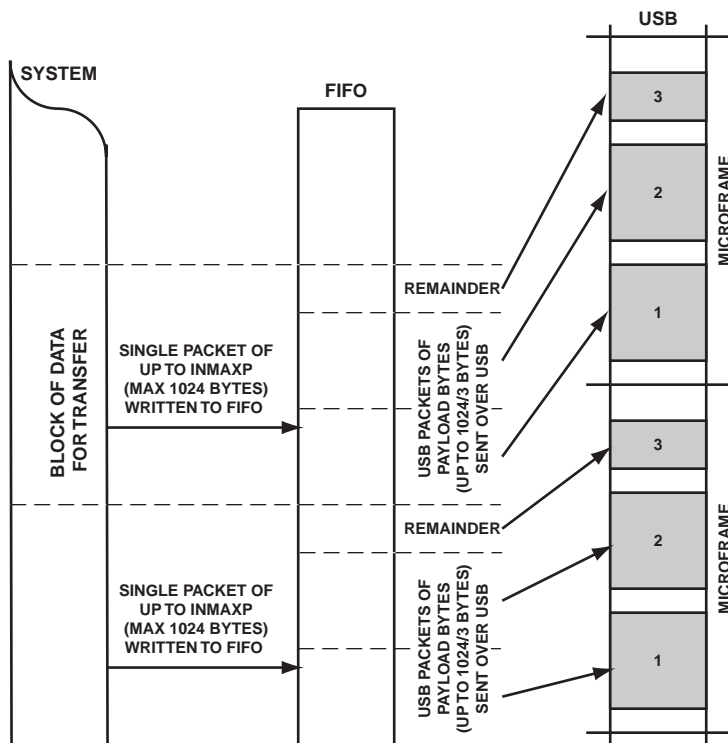


Figure 32-3. High-Bandwidth IN Endpoints

The USB controller supports these transfers by permitting the loading of data packets with up to 3 times the normal packet size into the associated FIFO in a single transaction. From the point of view of the software in the

processor core, the operation is then exactly as described above for single packet buffering or double packet buffering (as appropriate) *except* that `TXPKTRDY` always needs to be set manually (for example, set by the processor core) as the auto set feature does not operate with high-bandwidth isochronous transfers.

Any data packet loaded into the FIFO that is larger than the maximum payload is automatically split into USB packets of the maximum payload, or smaller, for transmission over the USB. The number of USB packets transmitted per micro-frame and the maximum payload in each packet is defined through the `USB_TX_MAX_PACKET` register. Bits 10–0 of the `USB_TX_MAX_PACKET` register determine the maximum payload in any USB packet while bits 12,11 determine the maximum number of such packets that can be sent in one micro-frame (2 or 3). Together, these set the maximum size of packet that can be loaded into the FIFO.

At least one USB packet always is sent. The number of further USB packets sent in the same micro-frame depends on the amount of data loaded into the FIFO. The `TXPKTRDY` bit is cleared and an interrupt is generated only when all the packets have been sent. Each USB packet is sent in response to an IN token. If, at the end of a micro-frame, the USB controller has not received enough IN tokens to send all the USB packets (for example, because one of the IN tokens received was corrupted), the remaining data is flushed from the FIFO. The `TXPKTRDY` bit is cleared and the `INCOMPTX_T` bit in the `USB_TXCSR` register is set to indicate that not all of the data loaded into the FIFO was sent.

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.

Description of Operation

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 32-1 on page 32-7](#)).

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.

High Bandwidth Isochronous OUT Endpoints

In high-speed mode, isochronous Rx endpoints can receive up to three USB packets in any micro-frame, with a payload of up to 1024/3 bytes in each packet, corresponding to a data transfer rate of up to 3072 bytes per micro-frame. [Figure 32-4](#) shows an overview of high-bandwidth OUT endpoints in USB.

The USB controller supports this rate by automatically combining all the USB packets received during a micro-frame into a single packet of up to 3 normal packets in size within the Rx FIFO. From the point of view of the software in the processor core, the operation is then exactly as described above for single packet buffering or double packet buffering (as appropriate), *except* the `RXPKTRDY` always needs to be cleared manually (for example, cleared by the processor core) as `AUTOCLEAR_R` does not operate with high-bandwidth isochronous transfers.

The maximum number of USB packets that may be received in any micro-frame and the maximum payload of these packets are defined through the `USB_RX_MAX_PACKET` register. Bits 10–0 of the `USB_RX_MAX_PACKET` register determine the maximum payload in any USB packet while bits 12,11 determine the maximum number of these packets that may be received in a micro-frame (2 or 3).

The number of USB packets sent in any micro-frame depends on the amount of data to be transferred, and is indicated through the PIDs used for the individual packets. If the indicated number of packets have not been received by the end of a micro-frame, the `INCOMPRX_R` bit in the `USB_RXCSR` register is set to indicate that the data in the FIFO is incomplete. Even so, an interrupt is still be generated to allow the data that is received to be read from the FIFO.

Description of Operation

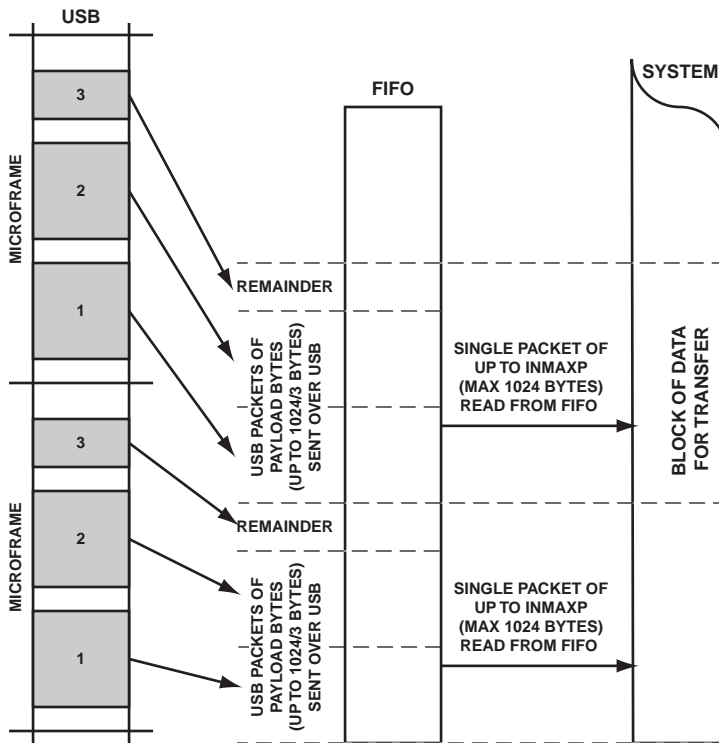


Figure 32-4. High-Bandwidth OUT Endpoints

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, which 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 `<USB_TX_MAX_PACKET` or `<USB_RX_MAX_PACKET`) to denote the end of a transfer.

Each of the workflows use the following common method:

1. Configure the endpoint (`USB_TX_MAX_PACKET` or `USB_RX_MAX_PACKET` value and control and status registers)
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).

In the workflow descriptions, processor core interactions are indicated with **bold face type**, and transfer parameters are indicated with *italic type*. Note that 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 32-141 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.

Description of Operation

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 transaction 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.

Note: The Setup packet associated with any Standard Device Request should include an 8-byte command. Any 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).

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.

NOTE, however, that 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.

Description of Operation

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 `SentStall` 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.

Peripheral Mode, Bulk IN, Transfer Size Known

Programming values must be known for the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes for this process:

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 the 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` automatically is 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

The programming value for the maximum individual packet size (*MaxPktSize*) in bytes is assumed to be an even number of bytes for this process:

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 and then **sets** `TXPKTRDY` or simply sends 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 dealt with by the ISR from step 4.

Description of Operation

Peripheral Mode, ISO IN, Small MaxPktSize

The programming value for the maximum individual packet size (*MaxPktSize*) in bytes is <128 bytes and is an even number of bytes, double buffering is assumed as enabled, and the auto set feature is unused (because packets are often <*MaxPktSize*) for this process:

1. **Load *MaxPktSize* into** USB_TX_MAX_PACKET
2. **Set** ISO_T=1 in USB_TXCSR
3. **Pre-load** 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

The programming value for the maximum individual packet size (*MaxPktSize*) in bytes is >128 bytes and is an even number of bytes, double buffering is assumed as enabled, and the auto set feature is unused (because packets are often <*MaxPktSize*) for this process:

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 in 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

Programming values must be known for the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes for this process:

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` automatically is cleared by the USB controller when each new packet is read.

Description of Operation

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

A programming value must be known for the maximum individual packet size (*MaxPktSize*) in bytes for this process:

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_DMAx-INT 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 **reading** it with the processor core.
6. On each USB_DMAxINT transition, the DMA controller reads a packet from the FIFO. RXPKTRDY automatically is 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 < *MaxPktSize*, the Rx interrupt goes high, and the ISR from step 5 reads out the remaining short packet.

Peripheral Mode, ISO OUT, Small MaxPktSize

The programming value for the maximum individual packet size (*MaxPktSize*) in bytes is <128 bytes, and double buffering is assumed as enabled for this process:

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 (equaling from the USB_RXCOUNT number of bytes) from the FIFO and 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

The programming value for the maximum individual packet size (*MaxPktSize*) in bytes is >128 bytes, and double buffering is assumed as enabled for this process:

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 or two packets (equaling from 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.

Description of Operation

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 microseconds. If no SOF packet is received after 1.00358 ms (or 125.125 ms), it is assumed that the packet is

lost and 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 ms) 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 this feature is enabled and the SOFT_CONN bit is zero, the PHY is put into non-driving and D+ and D– are three-stated. Then, the USB controller appears to have been disconnected to other devices on 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 into 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:

Description of Operation

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 request 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_CSRO` 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 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`.

Description of Operation

Host Mode Operation

Operations for the USB OTG interface when 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 32-13](#).

Control Transaction as a Host

Host Control Transactions are conducted through Endpoint 0 and 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).

As 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 which is in turn followed by an IN Status Phase.

Read Requests comprise a SETUP command, followed by an IN Data Phase which is in turn followed by an OUT Status Phase.

A timeout may be set to limit the length of time for which the USB controller will retry a transaction which is continually NAKed by the target. This limit can be between 2 and 2^{15} frames/microframes and is set through the USB_NAKLimit0 register.

The following sections describe the actions that the core needs to take in issuing these different types of request through looking at the steps to take in the different phases of a Control Transaction.

Note: Before initiating any 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 the target's new address.

Setup Phase as a Host

For the SETUP Phase of a Control Transaction, the PROCESSOR CORE driving the Host device needs to:

1. Load the 8 bytes of the required Device request command into the Endpoint 0 FIFO
2. Then set SETUPPKT_H and TxPTrdy (bits 3 and 1 of the USB_USB_CSR0 register, respectively). **Note:** These bits need to be set together.

The USB controller then proceeds to send a SETUP token followed by the 8-byte command to Endpoint 0 of the addressed device, retrying as necessary.

Description of Operation

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_CSRO` 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 any 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 ACKed 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

For the IN Data Phase of a Control Transaction, the PROCESSOR CORE driving the Host device needs to:

1. Set `REQPKT_H` in `USB_CSRO`.
2. Wait while the USB controller both sends the IN token and receives the required data back.

3. When the USB controller generates the Endpoint 0 interrupt (for example, sets `EPO_TX` in `USB_INTRTX` register), read `USB_CSRO` 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 any 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 Steps 1 – 4.

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)

For the `OUT Data Phase` of a `Control Transaction`, the `PROCESSOR CORE` driving the `Host device` needs to:

1. Load the data to be sent into the `Endpoint 0 FIFO`
2. Then set the `TxPktRdy` bit in `USB_CSRO`.

Description of Operation

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 `EPO_TX` in `USB_INTRTX` register). The PROCESSOR CORE should then read `USB_CSRO` 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 any 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` or `NAKLimit` is set, the OUT data is correctly ACKed.

4. If further data needs to be sent, the PROCESSOR CORE should repeat Steps 1 – 3.

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)

For the IN Status Phase of a Control Transaction, the PROCESSOR CORE driving the Host device needs to:

1. Set `STATUSPKT_H_H` and `REQPKT_H` (bits 6 and 5 of `USB_CSRO`, respectively). Note: These bits need to 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 `EPO_TX` in `USB_INTRTX` register), read `USB_CSRO` 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 any 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)

For the OUT Status Phase of a Control Transaction, the PROCESSOR CORE driving the Host device needs to:

Description of Operation

1. Set `STATUSPKT_H` and `TxPktRdy`. Note: These bits need to 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 any 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` or `NAK_TIMEOUT_H` is set, the STATUS Phase is correctly ACKed.

Host IN Transactions

When the USB controller is operating as a host, IN transactions are handled in much the same manner in which OUT transactions are handled when the USB controller is operating as a peripheral, except that the transaction needs first to 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 is operating as a host, OUT transactions are handled in a similar manner to the way IN transactions are handled when the USB controller is operating 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. Again, the `AUTOSET_T` bit can be used with an external DMA controller to perform complete bulk transfers without processor core intervention.

Description of Operation

If the target function responds to the OUT token with a NAK, the USB controller keeps retrying the transaction until any NAK limit that is 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 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 only is started if 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 also lets you to specify a limit to the length of time for NAKs may be received from a particular target before the endpoint is timed out (`USB_TXINTERVAL` or `USB_RXINTERVAL` registers).

Babble

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. 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, and generates a babble interrupt (`RESET_OR_BABLE_B`).

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.

Functional Description

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 only transfers
- Wait states are asserted when FIFO accesses take place (maximum of 3 are possible when contention for the SRAM occurs)

The DCB/DEB bus master interface has the following characteristics:

- 32-bit wide read and write data busses
- Write transfers of byte, half word and 32-bit words possible (byte and half word are used only for remaining bytes in a transfer)
- Read transfers of 32 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 32-2](#).

Table 32-2. 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 (>30MHz required) to generate an internal clock used to clock many of the system 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 24 MHz SCLK 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.

Programming Model

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 pullup 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.

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 requires to start a session, the processor core needs to 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), turns on VBUS, and waits for VBUS to go above the VBUS valid threshold, as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register to transition to 11.

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.

Programming Model

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 > 2 ms – the USB controller first pulses the data line then pulses VBUS (by taking high the interrupt `CHRG_VBUS_START` in `USB_OTG_VBUS_IRQ`).

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 32-122. 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 wishes for a session to start, 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 pulse 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 con-

troller detects this and clears the `SESSION` bit to indicate that the session has ended. A disconnect interrupt (`DISCON_B` bit in `USB_INTRUSB`) also is 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 becomes the host by setting the `HOST_REQ` bit in the `USB_OTG_DEV_CTL` register. This bit can be set either at the same time as 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), then assuming the `HOST_REQ` bit remains set, the controller enters host mode and begins host negotiation (as specified in the USB on-the-go 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.

Programming Model

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, including even 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

In order to conserve power when the chip is idle, systems often use various levels of powerdown modes which shut down power and clocks to various parts of the chip. Hibernate state is the highest power saving mode for the processor (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 is idle for a long enough period of time with no immediate need for the clocks to be active that the chip can be put into a power-down mode such as the hibernate state. As far as active USB sessions are concerned, this period of inactivity occurs when there is a USB suspend state (idle on

the bus for >3ms) or if no OTG session is valid. The `SUSPEND_MODE` bit (in `USB_POWER`) and `VBUS1-0` status bits (in `USB_OTG_DEV_CTL`) respectively are used to indicate these states.

Before the system software (driver) pushes processor into 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 PLL status (`PLL_STAT`) register. The `EXT_WAKE` pin can be used by the external power-up sequence chip to power up DDR or any other external peripheral. The processor typically goes through these steps (see [Figure 32-5](#)) when it comes out of hibernate state.

Programming Model

After the chip comes out of hibernate state, the software has to make sure that CSR_RSTD bit of the USB_APHY_CNTRL2 register is set. This setting deactivates the non-idle activity detection logic and ensure proper USB functionality.

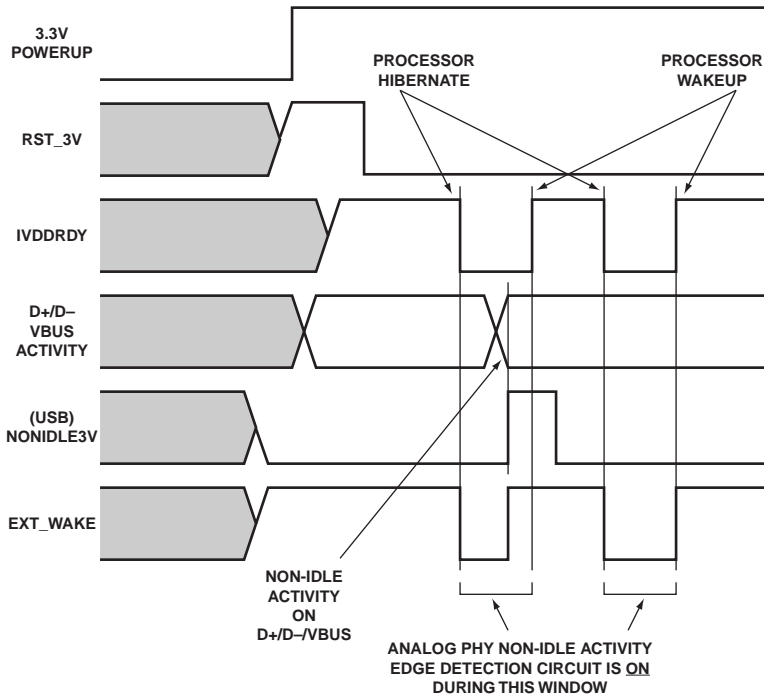


Figure 32-5. Timing Diagram of EXT_WAKE Pin

The interrupt will be asserted when either of the following events occur:

- Non-idle signaling occurs during USB suspend state (including USB reset signaling)
- VBUS falls below the session valid threshold

Wakeup without Re-Enumeration

When USB is kicked into suspend mode after 3 ms of inactivity on the D+ and D–, in order to save power, it is possible that the processor is pushed into hibernate state. Hibernate state implies all the internal power is shut down, and only the external 3.3 V power is present. Also, all the clocks in the processor are shut down. When USB is expected to wake up in response to non-idle activity on the D+ and D–, the USB controller has lost the state it was in before going into hibernate state. This lost state imposes a restriction on host to re-enumerate USB controller device. To prevent re-enumeration of USB controller device, systems must do the following:

- Before the system software (driver) pushes processor into hibernate state, the software must make sure that the state of the USB is stored in some 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 `csr_hbr1v` 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 states on 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 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 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.

Programming Model

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_rstd1v` 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 32-6) prevents re-enumeration of the USB controller device after the processor wakes up from hibernate state.

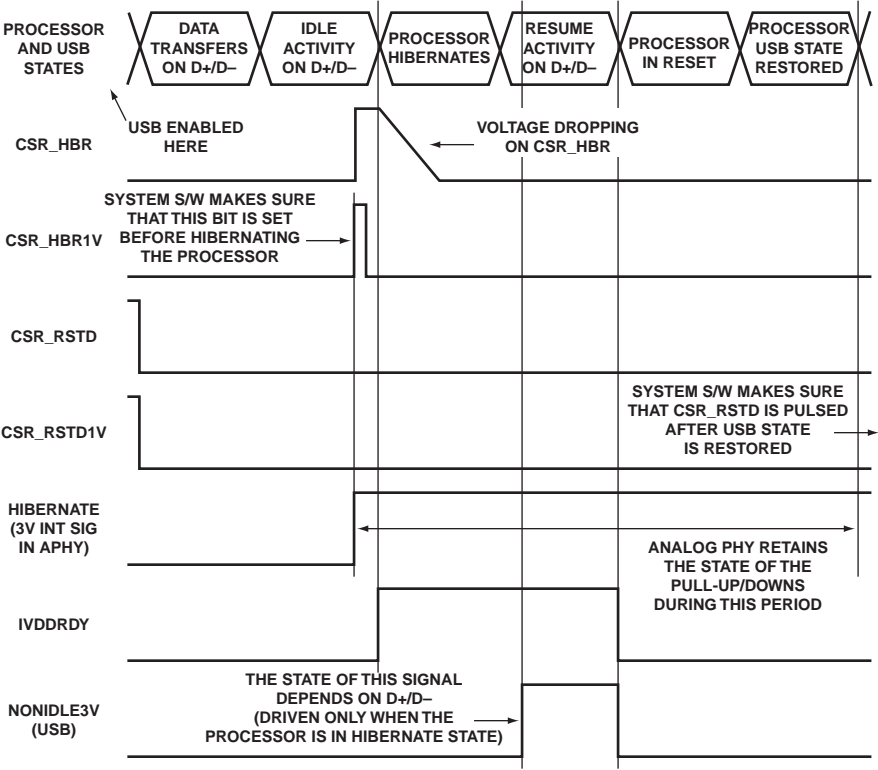


Figure 32-6. Timing Diagram of the CSR_HBR/CSR_RSTD Bits

Data Transfer

Regardless of whether the USB controller is operating in host or peripheral mode, data is channeled through the endpoint FIFOs in order to construct packets to be sent on USB or to receive packets from USB. The Rx FIFOs are used to receive OUT packets when in peripheral mode and

Programming Model

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 individual programmable options available to allow each to be set up individually. This is necessary because different transfer types need to 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 provides a means of 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 to prevent data loss or corruption. This problem 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 a byte remaining in the transfer, the USB controller latches the least significant byte of the last half word.

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.

Another use for the `USB_TXCOUNT` registers is to streamline DMA transfers, preventing unnecessary processor interaction in lengthy multi-packet transfers.

DMA Master Channels

The USB controller provides 8 DMA Master channels to provide a more efficient method of transferring larger amount of data between the FIFOs and 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.

Programming Model

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 USB. This mode can be used with any endpoint, either 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 packets of the transfer, 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.

DMA transfers may be 8-, 16-, or 32-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-, half word-, 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 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.

As long as the start address (written to the `DMAxADDR` register) is word aligned, all the transfers for a packet are word transfers (32 bits), with possible half-word and/or byte transfers added at the end to handle any residue. If the start address is merely half-word aligned, the DMA controller uses half-word transfers for the duration of the packet, with a possible byte transfer at the end. If the start address is an odd byte address, the DMA controller uses byte transfers for the duration of the packet.

Split transactions and retries are supported.

Transferring Packets Using DMA

Use of the DMA master channels to access the USB controller FIFOs requires both the appropriate channel and the endpoint to 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 are normally carried out using DMA mode 0.

For this, 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. (Note that 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

To carry out this operation using DMA mode 0, a USB controller TX endpoint is programmed as follow:

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. (Note that 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 transfer 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

The transfer of multiple packets normally are carried out using DMA mode 1. Where multiple packets are to be received 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`

Programming Model

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 `RXPKTTRDY` 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

For operation in DMA mode 1 with a USB controller 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. These registers are the:

- [USB Control Registers](#)
- [USB Packet Control – Indexed Registers](#)
- [USB Endpoint FIFO Registers](#)
- [USB OTG Control Registers](#)
- [USB PHY Control Registers](#)
- [USB Endpoint 0 Control Registers](#), [USB Endpoint 1 Control Registers](#), [USB Endpoint 2 Control Registers](#), [USB Endpoint 3 Control Registers](#), [USB Endpoint 4 Control Registers](#), [USB Endpoint 5 Control Registers](#), [USB Endpoint 6 Control Registers](#), [USB Endpoint 7 Control Registers](#)
- [USB DMA Registers](#), [USB Channel 0 Config Registers](#), [USB Channel 1 Config Registers](#), [USB Channel 2 Config Registers](#), [USB Channel 3 Config Registers](#), [USB Channel 4 Config Registers](#), [USB Channel 5 Config Registers](#), [USB Channel 6 Config Registers](#), [USB Channel 7 Config Registers](#)

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table 32-3. USB OTG Memory-Mapped Registers

Name	Width	Address	Index	Function
USB Control Registers				
USB_FADDR	16	0xFFC03C00	n/a	USB function address register on page 32-81
USB_POWER	16	0xFFC03C04	n/a	USB power management register on page 32-82
USB_INTRTX	16	0xFFC03C08	n/a	USB transmit interrupt register on page 32-85 for endpoint 0 and Tx endpoint 1 to 7
USB_INTRRX	16	0xFFC03C0C	n/a	USB receive interrupt register on page 32-86 for Rx endpoints 1 to 7
USB_INTRTXE	16	0xFFC03C10	n/a	USB transmit interrupt enable register on page 32-87 for IntrTx
USB_INTRRXE	16	0xFFC03C14	n/a	USB receive interrupt enable register on page 32-88 for IntrRx
USB_INTRUSB	16	0xFFC03C18	n/a	USB common interrupt register on page 32-89
USB_INTRUSBE	16	0xFFC03C1C	n/a	USB common interrupt enable register on page 32-90
USB_FRAME	16	0xFFC03C20	n/a	USB frame number register on page 32-91
USB_INDEX	16	0xFFC03C24	n/a	USB index register on page 32-91
USB_TESTMODE	16	0xFFC03C28	n/a	USB test mode register on page 32-93 (for Analog Devices internal use only)
USB_GLOBINTR	16	0xFFC03C2C	n/a	USB global interrupt register on page 32-94
USB_GLOBAL_CTL	16	0xFFC03C30	n/a	USB global control register on page 32-95

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Packet Control – Indexed Registers				
USB_TX_MAX_PACKET	16	0xFFC03C40	1–7	USB Tx maximum packet register on page 32-97
USB_CSR0	16	0xFFC03C44	0	USB control/status register on page 32-98
USB_TXCSR	16	0xFFC03C44	1–7	USB Tx control/status EPx register on page 32-102
USB_RX_MAX_PACKET	16	0xFFC03C48	1–7	USB Rxx maximum packet register on page 32-107
USB_RXCSR	16	0xFFC03C4C	1–7	USB Rx control/status EPx register on page 32-109 ^t
USB_COUNT0	16	0xFFC03C50	0	USB count 0 register on page 32-115
USB_RXCOUNT	16	0xFFC03C50	1–7	USB Rx byte count EPx register on page 32-116
USB_TXTYPE	16	0xFFC03C54	1–7	USB Tx type register on page 32-117
USB_NAKLIMIT0	16	0xFFC03C58	0	USB NAK 0 limit register on page 32-117
USB_TXINTERVAL	16	0xFFC03C58	1–7	USB Tx interval register on page 32-118
USB_RXTYPE	16	0xFFC03C5C	1–7	USB Rx type register on page 32-119
USB_RXINTERVAL	16	0xFFC03C60	1–7	USB Rx interval register on page 32-120
USB_TXCOUNT	16	0xFFC03C68	1–7	USB Tx byte count EPx register on page 32-121

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint FIFO Registers				
USB_EP0_FIFO	16	0xFFC03C80	n/a	USB endpoint FIFO 0 register on page 32-122
USB_EP1_FIFO	16	0xFFC03C88	n/a	USB endpoint FIFO 1 register on page 32-122
USB_EP2_FIFO	16	0xFFC03C90	n/a	USB endpoint FIFO 2 register on page 32-120
USB_EP3_FIFO	16	0xFFC03C98	n/a	USB endpoint FIFO 30 register on page 32-120
USB_EP4_FIFO	16	0xFFC03CA0	n/a	USB endpoint FIFO 4 register on page 32-120
USB_EP5_FIFO	16	0xFFC03CA8	n/a	USB endpoint FIFO 5 register on page 32-120
USB_EP6_FIFO	16	0xFFC03CB0	n/a	USB endpoint FIFO 6 register on page 32-120
USB_EP7_FIFO	16	0xFFC03CB8	n/a	USB endpoint FIFO 7 register on page 32-120
USB OTG Control Registers				
USB_OTG_DEV_CTL	16	0xFFC03D00	n/a	USB OTG device control register on page 32-122
USB_OTG_VBUS_IRQ	16	0xFFC03D04	n/a	USB OTG VBUS interrupt register on page 32-124
USB_OTG_VBUS_MASK	16	0xFFC03D08	n/a	USB VBUS mask register on page 32-126

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB PHY Control Registers				
USB_LINKINFO	16	0xFFC03D48	n/a	USB link info register on page 32-127
USB_VPLEN	16	0xFFC03D4C	n/a	USB VBUS pulse length register on page 32-127
USB_HS_EOF1	16	0xFFC03D50	n/a	USB high-speed EOF 1 register on page 32-128
USB_FS_EOF1	16	0xFFC03D54	n/a	USB full-speed EOF 1 register on page 32-128
USB_LS_EOF1	16	0xFFC03D58	n/a	USB low-speed EOF 1 register on page 32-129
USB_APHY_CNTRL	16	0xFFC03DE0	n/a	USB APHY control 2 register on page 32-130 <i>(for Analog Devices internal use only)</i>
USB_APHY_CALIB	16	0xFFC03DE4	n/a	USB APHY calibration register <i>(for Analog Devices internal use only)</i>
USB_APHY_CNTRL2	16	0xFFC03DE8	n/a	????? Used to prevent re-enumeration after the processor goes into hibernate mode
USB_PHY_TEST	16	0xFFC03DEC	n/a	Register used for PHY and FIFO test features <i>(for Analog Devices internal use only)</i>
USB_PLLOSC_CTRL	16	0xFFC03DF0	n/a	USB PLL OSC control register on page 32-132
USB_SRP_CLKDIV	16	0xFFC03DF4	n/a	USB SRP clock divider register on page 32-133

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 0 Control Registers				
USB_EP_NI0_TXMAXP	16	0xFFC03E00	n/a	Maximum packet size for host Tx endpoint0
USB_EP_NI0_TXCSR	16	0xFFC03E04	n/a	Control Status register for endpoint 0
USB_EP_NI0_RXMAXP	16	0xFFC03E08	n/a	Maximum packet size for host Rx endpoint0
USB_EP_NI0_RXCSR	16	0xFFC03E0C	n/a	Control Status register for host Rx endpoint0
USB_EP_NI0_RXCOUNT	16	0xFFC03E10	n/a	Number of bytes received in endpoint 0 FIFO
USB_EP_NI0_TXTYPE	16	0xFFC03E14	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint0
USB_EP_NI0_TXINTERVAL	16	0xFFC03E18	n/a	Sets the NAK response timeout on endpoint 0
USB_EP_NI0_RXTYPE	16	0xFFC03E1C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint0
USB_EP_NI0_RXINTERVAL	16	0xFFC03E20	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint0

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 1 Control Registers				
USB_EP_NI0_TXCOUNT	16	0xFFC03E28	n/a	Number of bytes to be written to the endpoint0 Tx FIFO
USB_EP_NI1_TXMAXP	16	0xFFC03E40	n/a	Maximum packet size for host Tx endpoint1
USB_EP_NI1_TXCSR	16	0xFFC03E44	n/a	Control Status register for endpoint1
USB_EP_NI1_RXMAXP	16	0xFFC03E48	n/a	Maximum packet size for host Rx endpoint1
USB_EP_NI1_RXCSR	16	0xFFC03E4C	n/a	Control Status register for host Rx endpoint1
USB_EP_NI1_RXCOUNT	16	0xFFC03E50	n/a	Number of bytes received in endpoint1 FIFO
USB_EP_NI1_TXTYPE	16	0xFFC03E54	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint1
USB_EP_NI1_TXINTERVAL	16	0xFFC03E58	n/a	Sets the NAK response timeout on endpoint1
USB_EP_NI1_RXTYPE	16	0xFFC03E5C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint1
USB_EP_NI1_RXINTERVAL	16	0xFFC03E60	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint1

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 2 Control Registers				
USB_EP_NI1_TXCOUNT	16	0xFFC03E68	n/a	Number of bytes to be written to the+H102 endpoint1 Tx FIFO
USB_EP_NI2_TXMAXP	16	0xFFC03E80	n/a	Maximum packet size for host Tx endpoint2
USB_EP_NI2_TXCSR	16	0xFFC03E84	n/a	Control Status register for endpoint2
USB_EP_NI2_RXMAXP	16	0xFFC03E88	n/a	Maximum packet size for host Rx endpoint2
USB_EP_NI2_RXCSR	16	0xFFC03E8C	n/a	Control Status register for host Rx endpoint2
USB_EP_NI2_RXCOUNT	16	0xFFC03E90	n/a	Number of bytes received in endpoint2 FIFO
USB_EP_NI2_TXTYPE	16	0xFFC03E94	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint2
USB_EP_NI2_TXINTERVAL	16	0xFFC03E98	n/a	Sets the NAK response timeout on endpoint2
USB_EP_NI2_RXTYPE	16	0xFFC03E9C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint2
USB_EP_NI2_RXINTERVAL	16	0xFFC03EA0	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint2

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 3 Control Registers				
USB_EP_NI2_ TXCOUNT	16	0xFFC03EA8	n/a	Number of bytes to be written to the endpoint2 Tx FIFO
USB_EP_NI3_ TXMAXP	16	0xFFC03EC0	n/a	Maximum packet size for host Tx endpoint3
USB_EP_NI3_ TXCSR	16	0xFFC03EC4	n/a	Control Status register for endpoint3
USB_EP_NI3_ RXMAXP	16	0xFFC03EC8	n/a	Maximum packet size for host Rx endpoint3
USB_EP_NI3_ RXCSR	16	0xFFC03ECC	n/a	Control Status register for host Rx endpoint3
USB_EP_NI3_ RXCOUNT	16	0xFFC03ED0	n/a	Number of bytes received in endpoint3 FIFO
USB_EP_NI3_ TXTYPE	16	0xFFC03ED4	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint3
USB_EP_NI3_ _TXINTERVAL	16	0xFFC03ED8	n/a	Sets the NAK response timeout on endpoint3
USB_EP_NI3_ RXTYPE	16	0xFFC03EDC	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint3
USB_EP_NI3_ _RXINTERVAL	16	0xFFC03EE0	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint3

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 4 Control Registers				
USB_EP_NI3_TXCOUNT	16	0xFFC03EE8	n/a	Number of bytes to be written to the H124endpoint3 Tx FIFO
USB_EP_NI4_TXMAXP	16	0xFFC03F00	n/a	Maximum packet size for host Tx endpoint4
USB_EP_NI4_TXCSR	16	0xFFC03F04	n/a	Control Status register for endpoint4
USB_EP_NI4_RXMAXP	16	0xFFC03F08	n/a	Maximum packet size for host Rx endpoint4
USB_EP_NI4_RXCSR	16	0xFFC03F0C	n/a	Control Status register for host Rx endpoint4
USB_EP_NI4_RXCOUNT	16	0xFFC03F10	n/a	Number of bytes received in endpoint4 FIFO
USB_EP_NI4_TXTYPE	16	0xFFC03F14	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint4
USB_EP_NI4_TXINTERVAL	16	0xFFC03F18	n/a	Sets the NAK response timeout on endpoint4
USB_EP_NI4_RXTYPE	16	0xFFC03F1C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint4
USB_EP_NI4_RXINTERVAL	16	0xFFC03F20	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint4

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 5 Control Registers				
USB_EP_NI4_TXCOUNT	16	0xFFC03F28	n/a	Number of bytes to be written to the endpoint4 Tx FIFO
USB_EP_NI5_TXMAXP	16	0xFFC03F40	n/a	Maximum packet size for host Tx endpoint5
USB_EP_NI5_TXCSR	16	0xFFC03F44	n/a	Control Status register for endpoint5
USB_EP_NI5_RXMAXP	16	0xFFC03F48	n/a	Maximum packet size for host Rx endpoint5
USB_EP_NI5_RXCSR	16	0xFFC03F4C	n/a	Control Status register for host Rx endpoint5
USB_EP_NI5_RXCOUNT	16	0xFFC03F50	n/a	Number of bytes received in endpoint5 FIFO
USB_EP_NI5_TXTYPE	16	0xFFC03F54	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint5
USB_EP_NI5_TXINTERVAL	16	0xFFC03F58	n/a	Sets the NAK response timeout on endpoint5
USB_EP_NI5_RXTYPE	16	0xFFC03F5C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint5
USB_EP_NI5_RXINTERVAL	16	0xFFC03F60	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint5

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 6 Control Registers				
USB_EP_NI5_TXCOUNT	16	0xFFC03F68	n/a	Number of bytes to be written to the H145endpoint5 Tx FIFO
USB_EP_NI6_TXMAXP	16	0xFFC03F80	n/a	Maximum packet size for host Tx endpoint6
USB_EP_NI6_TXCSR	16	0xFFC03F84	n/a	Control Status register for endpoint6
USB_EP_NI6_RXMAXP	16	0xFFC03F88	n/a	Maximum packet size for host Rx endpoint6
USB_EP_NI6_RXCSR	16	0xFFC03F8C	n/a	Control Status register for host Rx endpoint6
USB_EP_NI6_RXCOUNT	16	0xFFC03F90	n/a	Number of bytes received in endpoint6 FIFO
USB_EP_NI6_TXTYPE	16	0xFFC03F94	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint6
USB_EP_NI6_TXINTERVAL	16	0xFFC03F98	n/a	Sets the NAK response timeout on endpoint6
USB_EP_NI6_RXTYPE	16	0xFFC03F9C	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint6
USB_EP_NI6_RXINTERVAL	16	0xFFC03FA0	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint6

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Endpoint 7 Control Registers				
USB_EP_NI6_ TXCOUNT	16	0xFFC03FA8	n/a	Number of bytes to be written to the endpoint6 Tx FIFO
USB_EP_NI7_ TXMAXP	16	0xFFC03FC0	n/a	Maximum packet size for host Tx endpoint7
USB_EP_NI7_ TXCSR	16	0xFFC03FC4	n/a	Control Status register for endpoint7
USB_EP_NI7_ RXMAXP	16	0xFFC03FC8	n/a	Maximum packet size for host Rx endpoint7
USB_EP_NI7_ RXCSR	16	0xFFC03FCC	n/a	Control Status register for host Rx endpoint7
USB_EP_NI7_ RXCOUNT	16	0xFFC03FD0	n/a	Number of bytes received in endpoint7 FIFO
USB_EP_NI7_ TXTYPE	16	0xFFC03FD4	n/a	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint7
USB_EP_NI7_ _TXINTERVAL	16	0xFFC03FD8	n/a	Sets the NAK response timeout on endpoint7
USB_EP_NI7_ RXTYPE	16	0xFFC03FDC	n/a	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint7
USB_EP_NI7_ _RXINTERVAL	16	0xFFC03FF0	n/a	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint7
USB_EP_NI7_ TXCOUNT	16	0xFFC03FF8	n/a	Number of bytes to be written to the endpoint7 Tx FIFO
USB DMA Registers				
USB_DMA_ INTERRUPT	16	0xFFC04000	n/a	USB DMA interrupt register on page 32-134

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Channel 0 Config Registers				
USB_DMA0CONTROL	16	0xFFC04004	n/a	USB DMA control register 0 on page 32-135
USB_DMA0ADDR LOW	16	0xFFC04008	n/a	USB DMA address low register 0 on page 32-137
USB_DMA0ADDR HIGH	16	0xFFC0400C	n/a	USB DMA address high register 0 on page 32-138
USB_DMA0COUNT LOW	16	0xFFC04010	n/a	USB DMA count low register 0 on page 32-139
USB_DMA0COUNT HIGH	16	0xFFC04014	n/a	USB DMA count high register 0 on page 32-140
USB Channel 1 Config Registers				
USB_DMA1CONTROL	16	0xFFC04024	n/a	USB DMA control register 1 on page 32-135
USB_DMA1ADDR LOW	16	0xFFC04028	n/a	USB DMA address low register 1 on page 32-137
USB_DMA1ADDR HIGH	16	0xFFC0402C	n/a	USB DMA address high register 1 on page 32-138
USB_DMA1COUNT LOW	16	0xFFC04030	n/a	USB DMA count low register 1 on page 32-139
USB_DMA1COUNT HIGH	16	0xFFC04034	n/a	USB DMA count high register 1 on page 32-140

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Channel 2 Config Registers				
USB_DMA2CONTROL	16	0xFFC04044	n/a	USB DMA control register 2 on page 32-135
USB_DMA2ADDR LOW	16	0xFFC04048	n/a	USB DMA address low register 2 on page 32-137
USB_DMA2ADDR HIGH	16	0xFFC0404C	n/a	USB DMA address high register 2 on page 32-138
USB_DMA2COUNT LOW	16	0xFFC04050	n/a	USB DMA count low register 2 on page 32-139
USB_DMA2COUNT HIGH	16	0xFFC04054	n/a	USB DMA count high register 2 on page 32-140
USB Channel 3 Config Registers				
USB_DMA3CONTROL	16	0xFFC04064	n/a	USB DMA control register 3 on page 32-135
USB_DMA3ADDR LOW	16	0xFFC04068	n/a	USB DMA address low register 3 on page 32-137
USB_DMA3ADDR HIGH	16	0xFFC0406C	n/a	USB DMA address high register 3 on page 32-138
USB_DMA3COUNT LOW	16	0xFFC04070	n/a	USB DMA count low register 3 on page 32-139
USB_DMA3COUNT HIGH	16	0xFFC04074	n/a	USB DMA count high register 3 on page 32-140

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Channel 4 Config Registers				
USB_DMA4CONTROL	16	0xFFC04084	n/a	USB DMA control register 4 on page 32-135
USB_DMA4ADDR LOW	16	0xFFC04088	n/a	USB DMA address low register 4 on page 32-137
USB_DMA4ADDR HIGH	16	0xFFC0408C	n/a	USB DMA address high register 4 on page 32-138
USB_DMA4COUNT LOW	16	0xFFC04090	n/a	USB DMA count low register 4 on page 32-139
USB_DMA4COUNT HIGH	16	0xFFC04094	n/a	USB DMA count high register 4 on page 32-140
USB Channel 5 Config Registers				
USB_DMA5CONTROL	16	0xFFC040A4	n/a	USB DMA control register 5 on page 32-135
USB_DMA5ADDR LOW	16	0xFFC040A8	n/a	USB DMA address low register 5 on page 32-137
USB_DMA5ADDR HIGH	16	0xFFC040AC	n/a	USB DMA address high register 5 on page 32-138
USB_DMA5COUNT LOW	16	0xFFC040B0	n/a	USB DMA count low register 5 on page 32-139
USB_DMA5COUNT HIGH	16	0xFFC040B4	n/a	USB DMA count high register 5 on page 32-140

USB OTG Registers

Table 32-3. USB OTG Memory-Mapped Registers (Cont'd)

Name	Width	Address	Index	Function
USB Channel 6 Config Registers				
USB_DMA6CONTROL	16	0xFFC040C4	n/a	USB DMA control register 6 on page 32-135
USB_DMA6ADDR LOW	16	0xFFC040C8	n/a	USB DMA address low register 6 on page 32-137
USB_DMA6ADDR HIGH	16	0xFFC040CC	n/a	USB DMA address high register 6 on page 32-138
USB_DMA6COUNT LOW	16	0xFFC040D0	n/a	USB DMA count low register 6 on page 32-139
USB_DMA6COUNT HIGH	16	0xFFC040D4	n/a	USB DMA count high register 6 on page 32-140
USB Channel 7 Config Registers				
USB_DMA7CONTROL	16	0xFFC040E4	n/a	USB DMA control register 7 on page 32-135
USB_DMA7ADDR LOW	16	0xFFC040E8	n/a	USB DMA address low register 7 on page 32-137
USB_DMA7ADDR HIGH	16	0xFFC040EC	n/a	USB DMA address high register 7 on page 32-138
USB_DMA7COUNT LOW	16	0xFFC040F0	n/a	USB DMA count low register 7 on page 32-139
USB_DMA7COUNT HIGH	16	0xFFC040F4	n/a	USB DMA count high register 7 on page 32-140

USB Function Address (USB_FADDR) Register

The USB_FADDR register (see [Figure 32-7](#)) contains the 7-bit address of the peripheral part of the transaction.

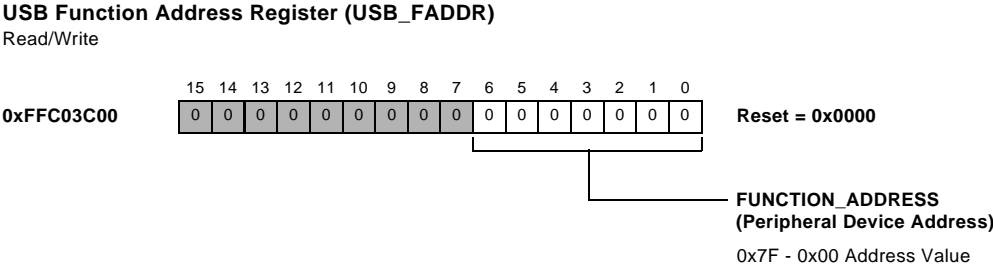


Figure 32-7. USB Function Address (USB_FADDR) Register

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 Power Management (USB_POWER) Register

The `USB_POWER` register (see [Figure 32-8](#)) controls suspend and resume signaling and controls some operational aspects of the USB controller.

USB Power Management Register (USB_POWER)

Read/Write, Read Only

Host Mode Access ro ro ro ro ro ro ro ro - - r/w ro r/w r/w set r/w
 Per. Mode Access ro ro ro ro ro ro ro ro r/w r/w r/w ro ro r/w ro r/w

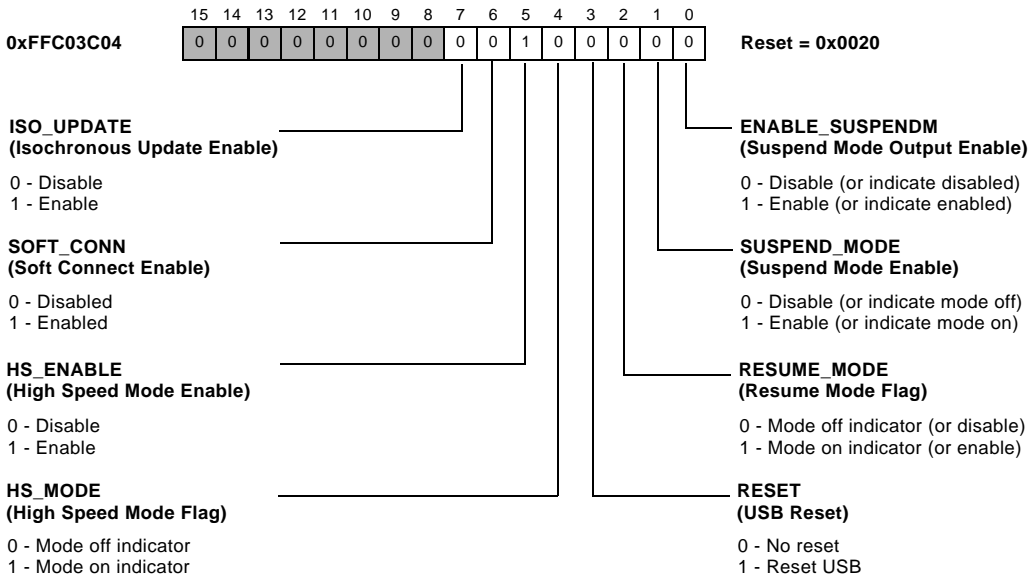


Figure 32-8. USB Power Management (USB_POWER) Register

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, the `SUSPEND_MODE` (bit 1) bit 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.

USB OTG Registers

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 Transmit Interrupt (USB_INTRTX) Register

The USB_INTRTX register (see Figure 32-9) 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.

USB Transmit Interrupt Register (USB_INTRTX)

Read/Write

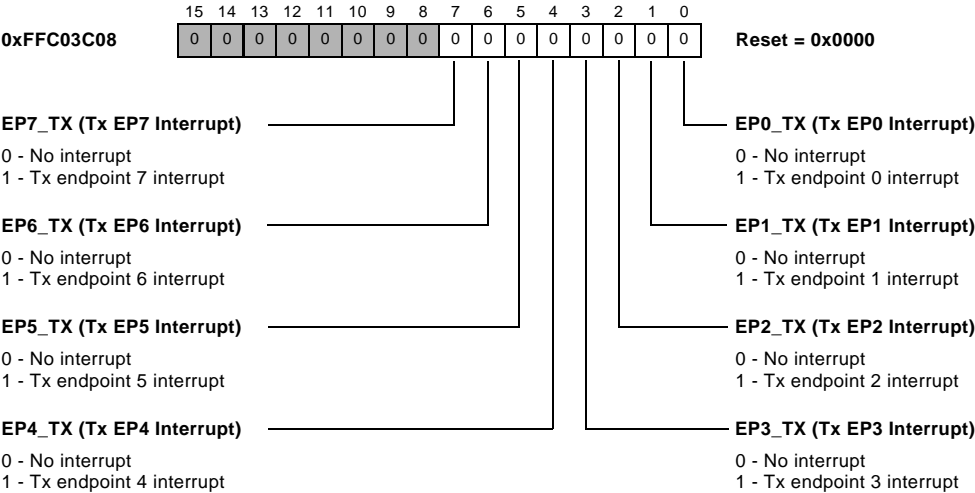


Figure 32-9. USB Transmit Interrupt (USB_INTRTX) Register

USB Receive Interrupt (USB_INTRRX) Register

The USB_INTRRX register (see [Figure 32-10](#)) 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.

USB Receive Interrupt Register (USB_INTRRX)

Read/Write

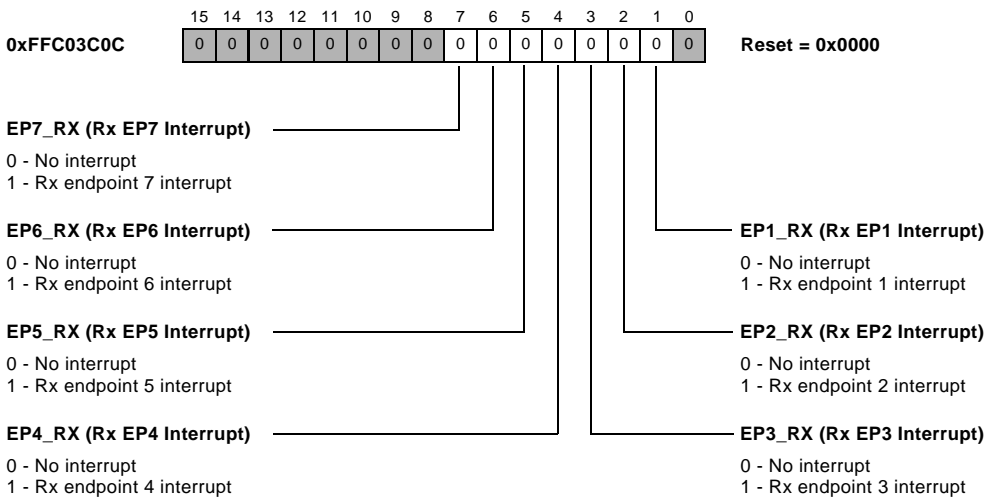


Figure 32-10. USB Receive Interrupt (USB_INTRRX) Register

USB Transmit Interrupt Enable (USB_INTRTXE) Register

The USB_INTRTXE register (see [Figure 32-11](#)) enables interrupts for endpoint 0 and the Tx endpoints 1–7.

USB Transmit Interrupt Enable Register (USB_INTRTXE)

Read/Write

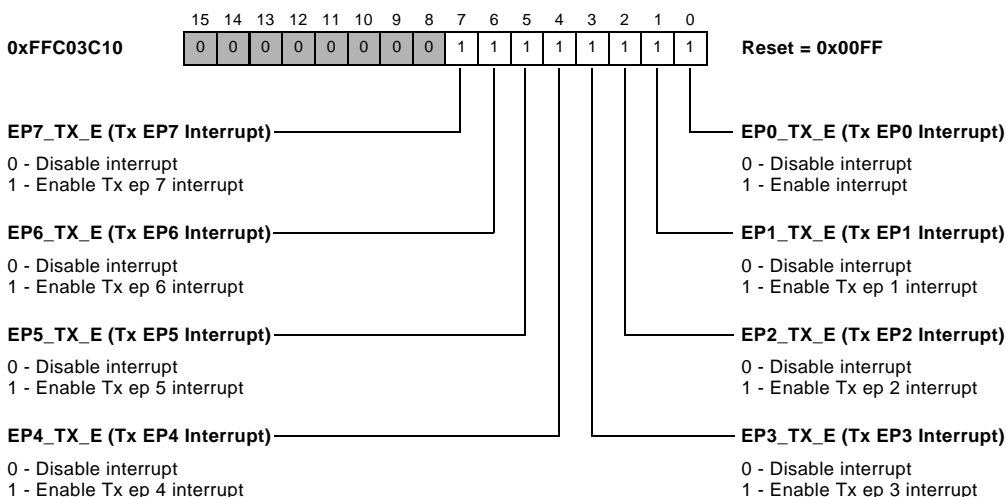


Figure 32-11. USB Transmit Interrupt Enable (USB_INTRTXE) Register

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 32-12](#)) enables interrupts for the Rx endpoints 1–7.

USB Receive Interrupt Enable Register (USB_INTRRXE)

Read/Write

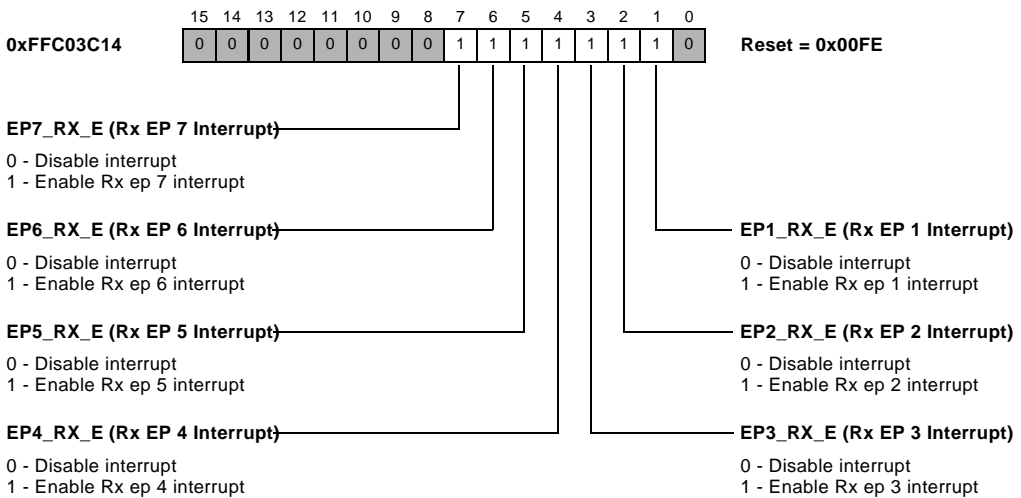


Figure 32-12. USB Receive Interrupt Enable (USB_INTRRXE) Register

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 32-13](#)) indicates which USB interrupts are currently active.

USB Common Interrupts Register (USB_INTRUSB)

Read/Write

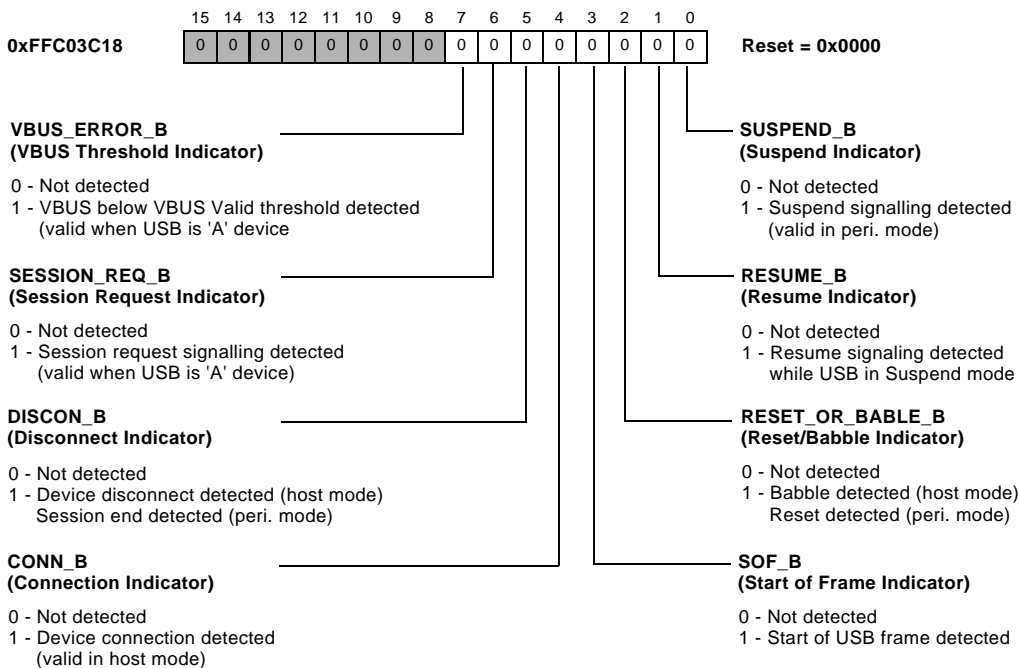


Figure 32-13. USB Common Interrupts (USB_INTRUSB) Register

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 32-14](#)) 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_BABBLE_BE` bits are set to 1 (for example, interrupts for resume signalling detection and reset/babble detection are enabled).

USB Common Interrupts Enable Register (USB_INTRUSBE)

Read/Write

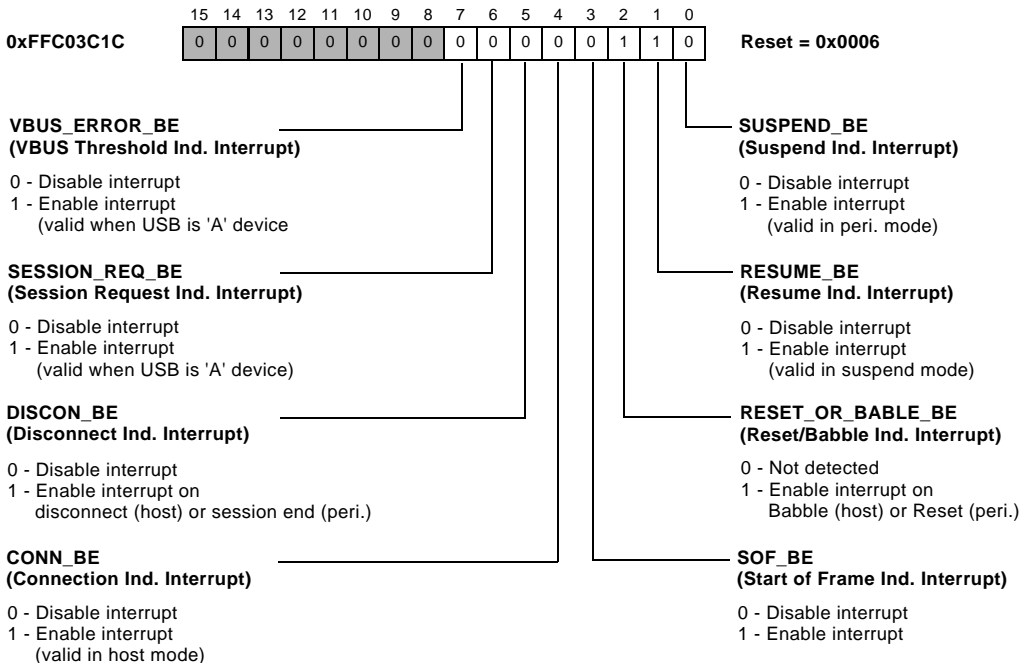


Figure 32-14. USB Common Int. Enable (USB_INTRUSBE) Register

USB Frame Number (USB_FRAME) Register

The USB_FRAME register (see [Figure 32-15](#)) contains the last received frame number; bit 10 is MSB; bit 0 is LSB.

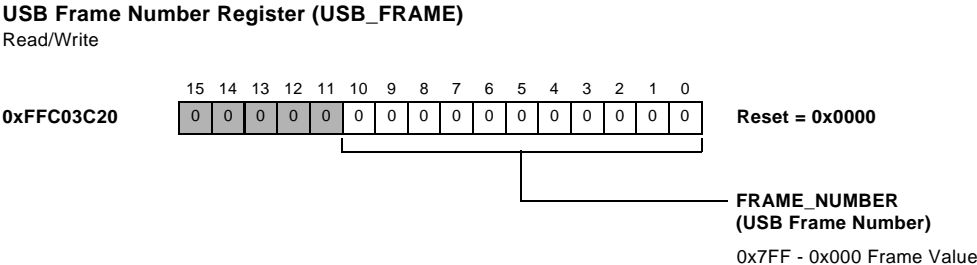


Figure 32-15. USB Frame Number (USB_FRAME) Register

USB Index (USB_INDEX) Register

The USB_INDEX register (see [Figure 32-16](#)) contains an index value for alternate addressing of USB endpoint control and status registers.

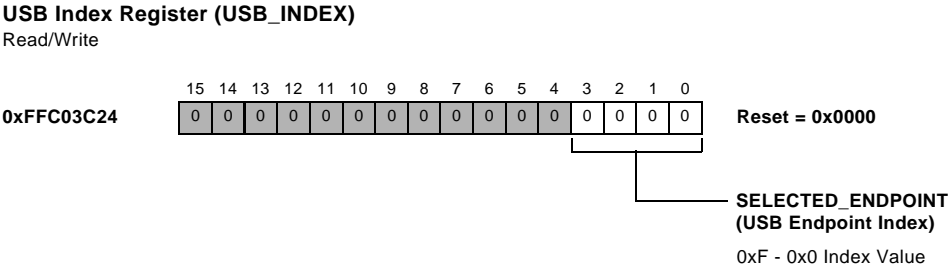


Figure 32-16. USB Index (USB_INDEX) Register

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 con-

USB OTG Registers

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 Test Mode (USB_TESTMODE) Register

The USB_TESTMODE register (see Figure 32-17) 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

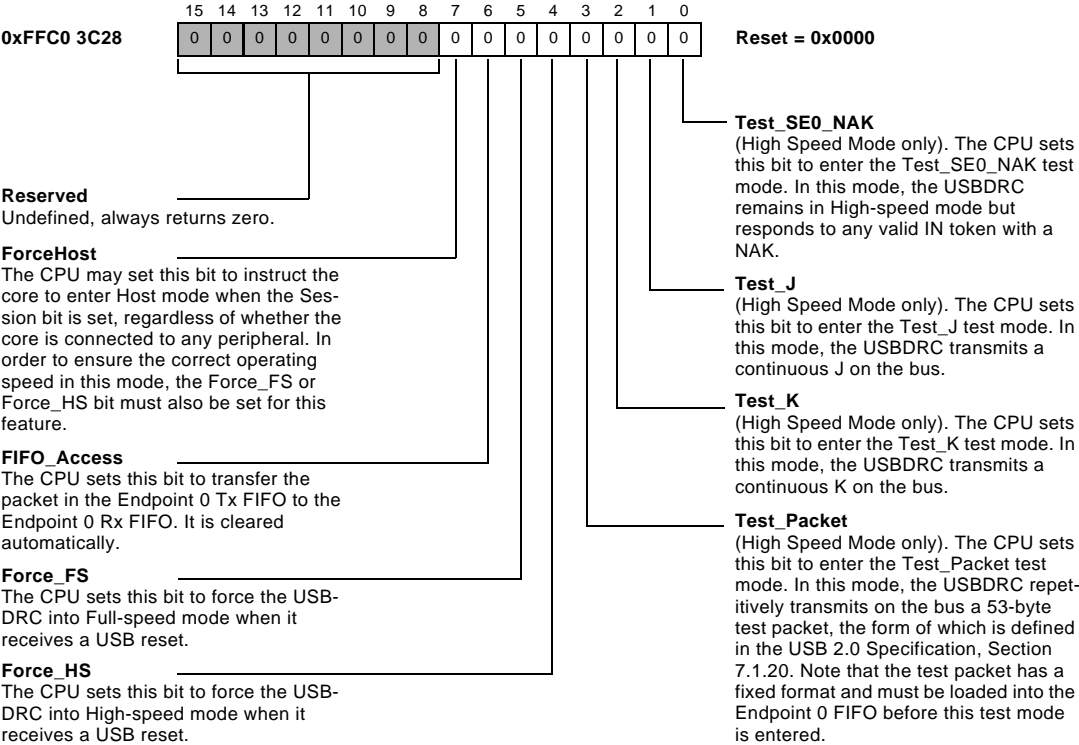


Figure 32-17. USB Test Mode (USB_TESTMODE) Register

USB_TESTMODE is not used in normal operation. Only one of the bits may be set at any one time, except for bit 5 in conjunction with the ForceHost feature.

USB Global Interrupt (USB_GLOBINTR) Register

The USB_GLOBINTR register (see [Figure 32-18](#)) 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).

USB Global Interrupt Register (USB_GLOBINTR)

Read/Write

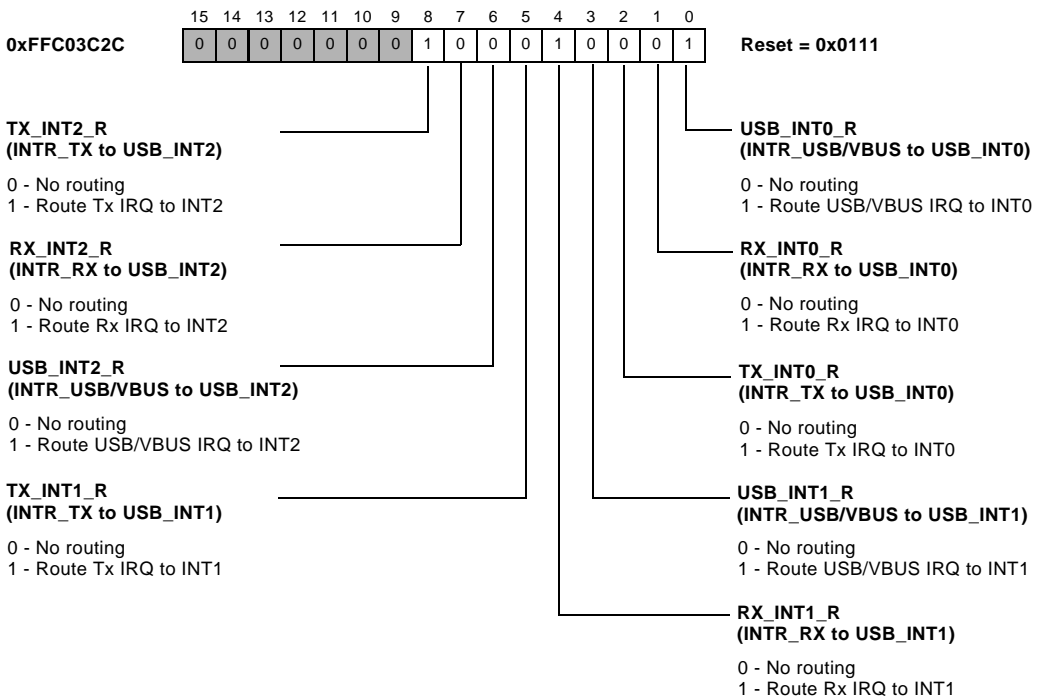


Figure 32-18. USB Global Interrupt (USB_GLOBINTR) Register

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 Global Control (USB_GLOBAL_CTL) Register

The USB_GLOBAL_CTL register (see [Figure 32-19](#)) 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.

USB Global Control Register (USB_GLOBAL_CTL)

Read/Write

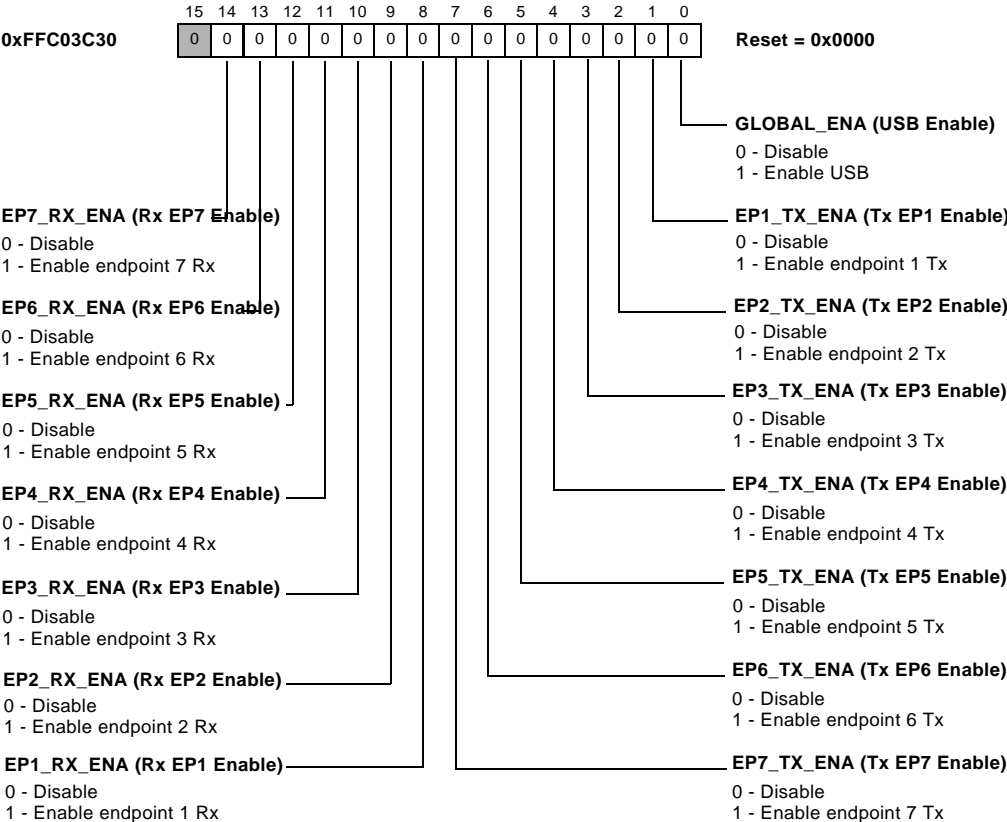



Figure 32-19. USB Global Control (USB_GLOBAL_CTL) Register

USB OTG Registers

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 any 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 32-45](#).

 Bit 15, which is marked as reserved in [Figure 32-19](#), 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 Tx Max Packet (USB_TX_MAX_PACKET) Register

The `USB_TX_MAX_PACKET` register (see [Figure 32-20](#)) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame.

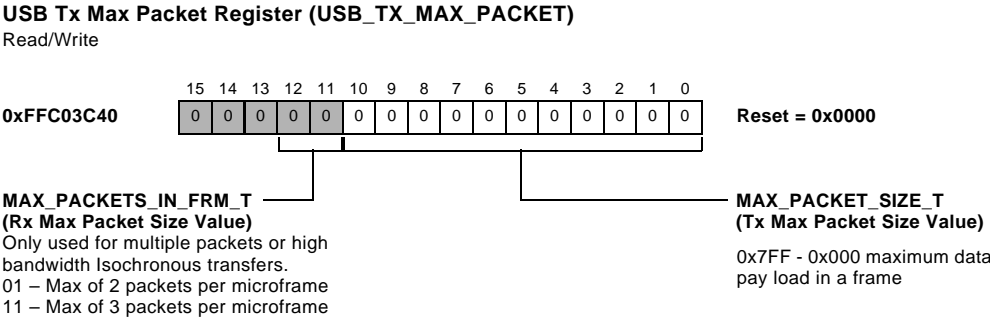


Figure 32-20. USB Tx Max Packet (USB_TX_MAX_PACKET) Register

When setting this value, you must note 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).

USB Control/Status EP0 (USB_CSR0) Register

The USB_CSR0 register (see [Figure 32-21](#)) provides control and status bits for endpoint 0. The *cleared automatically* notes in the figure indicate the self-clearing properties of that particular bit. The interpretation of the USB_CSR0 register depends on whether the USB controller is acting as a peripheral or as a host.

USB Control/Status EP0 Register (USB_CSR0)

Read/Write

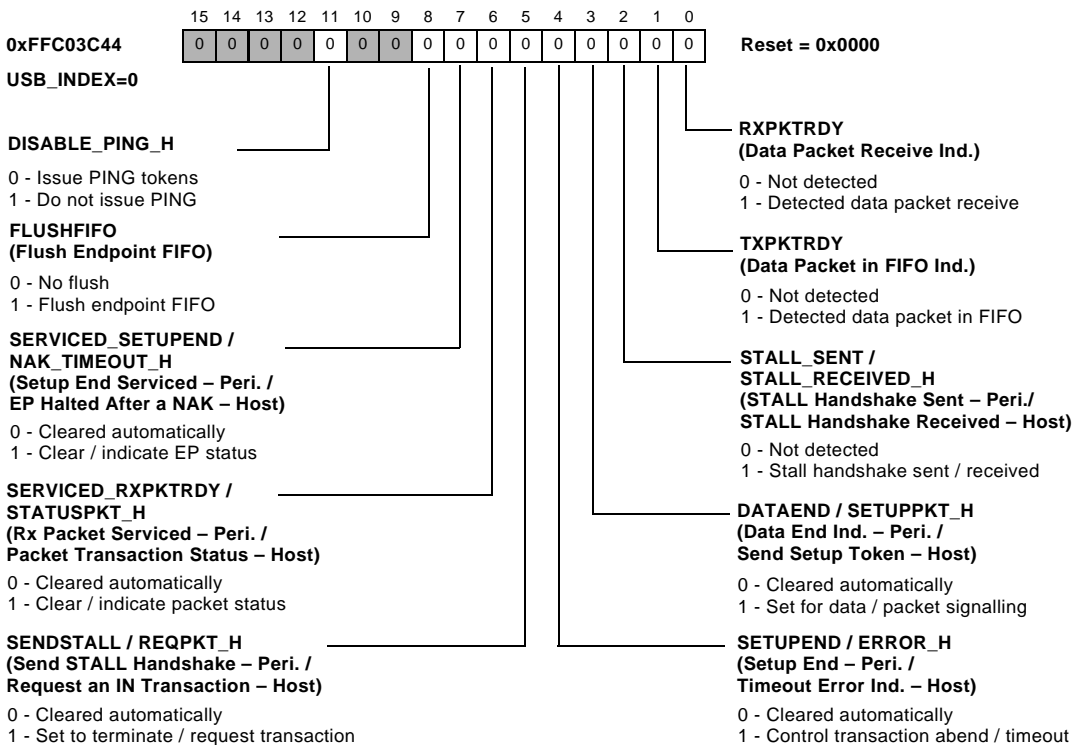


Figure 32-21. USB Control/Status EP0 (USB_CSR0) Register

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:

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 `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. Note that `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. Note that `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 32-22) provides control and status bits for transfers through the currently-selected Tx endpoint.

USB Tx Control/Status EPx Register (USB_TXCSR)

Read/Write

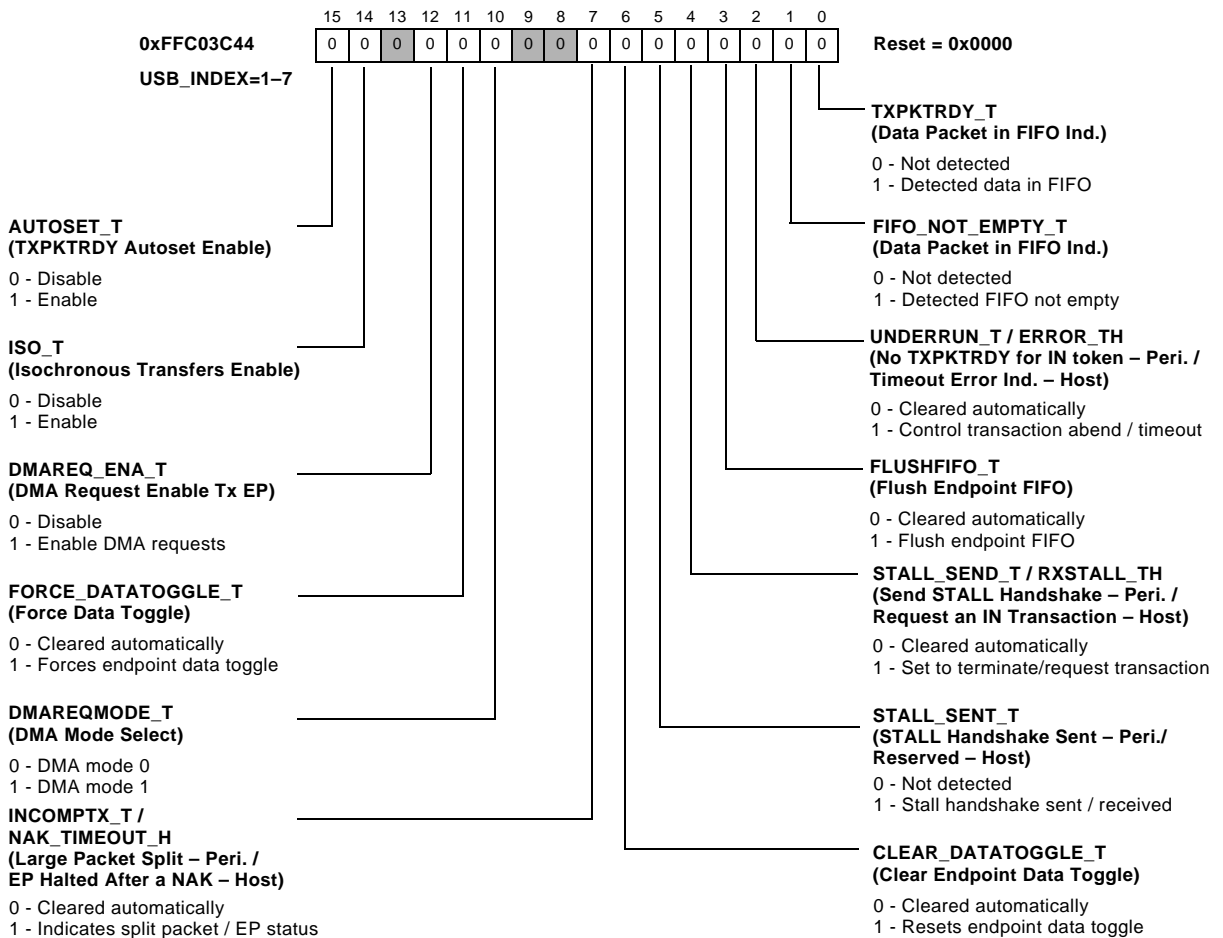


Figure 32-22. USB Tx Control/Status EPx (USB_TXCSR) Register

The *cleared automatically* notes in the figure indicate the self-clearing properties of that particular bit. 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 registers 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 1 packet in the Tx FIFO.

In host mode, the USB sets `FIFO_NOT_EMPTY_T` (bit 1) when there is at least 1 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.

USB OTG Registers

In host mode, the USB sets `ERROR_TH` (bit 2) when 3 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. Note that `FLUSHFIFO_T` has no effect unless `TXPKTRDY` is set. Also 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. Note that `FLUSHFIFO_T` has no effect unless `TXPKTRDY` is set. Also 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. Note: 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, when the endpoint is being used for high-bandwidth isochronous/interrupt transfers, `INCOMPTX_T` (bit 7) is set to indicate when a large packet is split into 2 or 3 packets for transmission but insufficient IN tokens have been received to send all the parts. Note: In anything other than a high-bandwidth transfer, 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. Note: 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. Note: This bit only has any 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.
 Note: This bit should not be set for high-bandwidth isochronous endpoints.

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. Note: should not be set for high-bandwidth isochronous endpoints.

USB Rx Max Packet (USB_RX_MAX_PACKET) Register

The `USB_RX_MAX_PACKET` register (see [Figure 32-23](#)) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame.

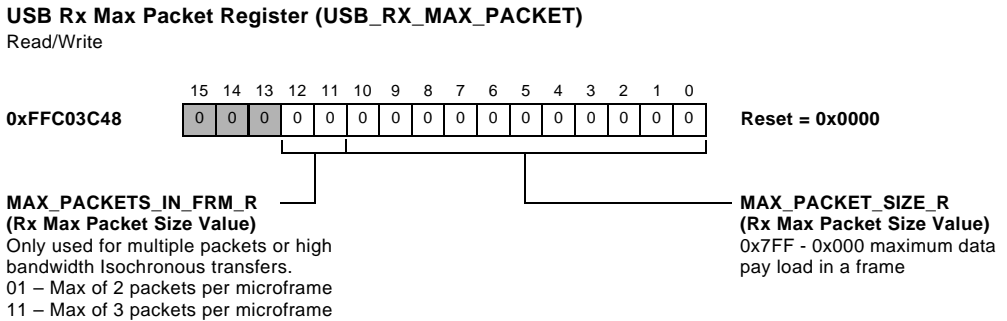



Figure 32-23. USB Rx Max Packet (USB_RX_MAX_PACKET) Register

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.

USB OTG Registers

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 32-24](#)) provides control and status bits for transfers through the currently-selected Rx endpoint.

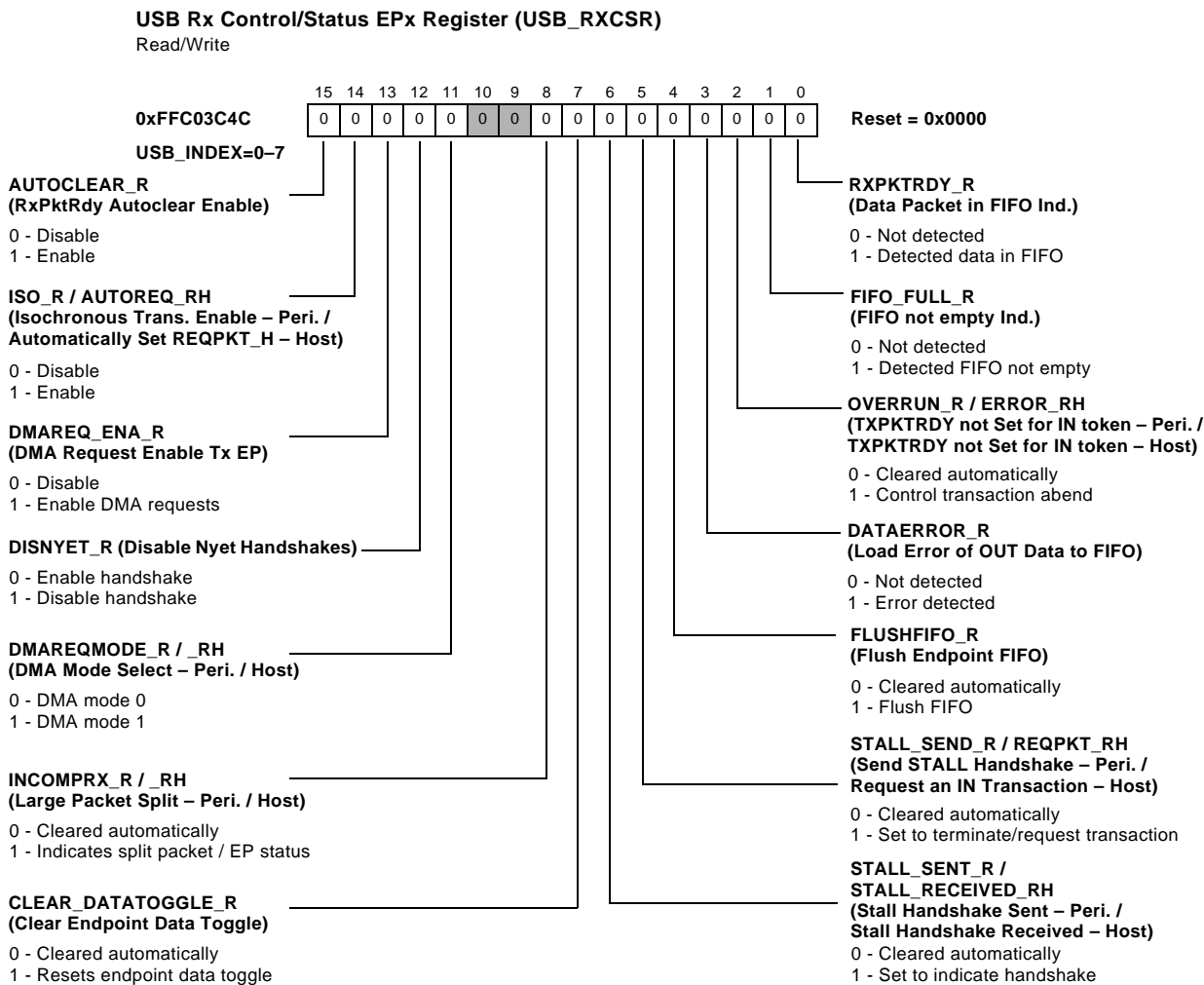


Figure 32-24. USB Rx Control/Status EPx (USB_RXCSR) Register

USB OTG Registers

The *cleared automatically* notes in the figure indicate the self-clearing properties of that particular bit. 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 registers 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. Note: 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. Note: 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. Note: 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. Note that `FLUSHFIFO_R` has no effect unless `RXPKTRDY` is set. Also 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. Note: `FLUSHFIFO_R` has no effect unless `RXPKTRDY` is set. Also 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. Note: 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) is set in a high-bandwidth isochronous transfer if the packet in the Rx FIFO is incomplete because parts of the data were not received. It is cleared when the RXPKTRDY is cleared.

Note: In anything other than a high-bandwidth isochronous transfer, this bit always returns 0.

In host mode, INCOMPRX_RH (bit 8) is set in a high-bandwidth isochronous transfer if the packet received is incomplete. It is cleared when the RXPKTRDY is cleared. Note: If USB protocols are followed correctly, this bit

should never be set. When it gets set, it indicates a failure of the associated peripheral device to operate within the USB specification. In anything other than a high-bandwidth isochronous transfer, this bit 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. Note: This bit only has any effect in high-speed mode, in which mode 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. Note: This bit only has any effect in high-speed mode, when 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.

USB OTG Registers

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. Note: Should not be set for high-bandwidth endpoints.

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. Note: Should not be set for high-bandwidth isochronous endpoints.

USB Count 0 (USB_COUNT0) Register

The USB_COUNT0 register (see [Figure 32-25](#)) 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 RXPKT_RDY is set.

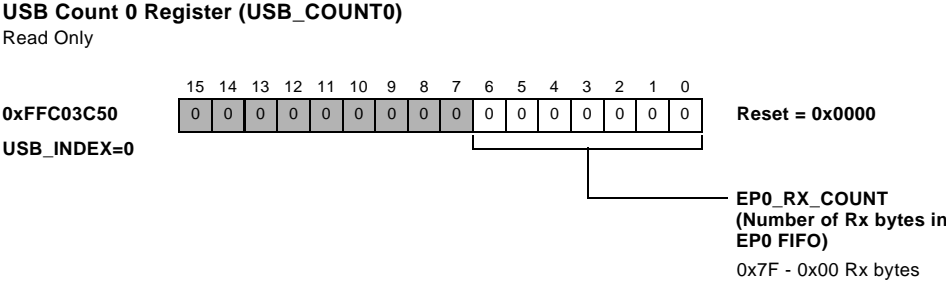


Figure 32-25. USB Count 0 (USB_COUNT0) Register

USB Rx Byte Count EPx (USB_RXCOUNT) Register

The USB_RXCOUNT register (see [Figure 32-26](#)) holds the number of received data bytes in the packet in the Rx FIFO. Note: 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

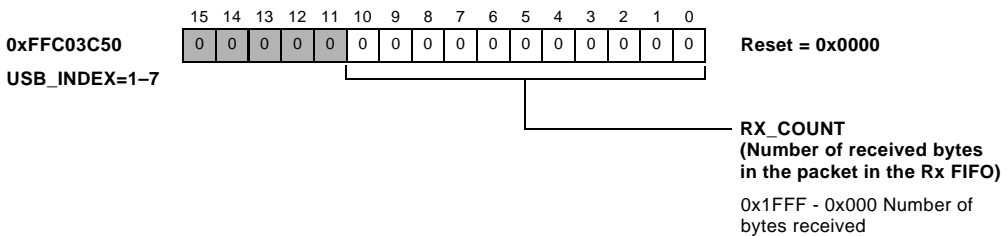


Figure 32-26. USB Rx Byte Count (USB_RXCOUNT) Register

USB Tx Type (USB_TXTYPE) Register

The `USB_TXTYPE` register (see [Figure 32-27](#)) 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

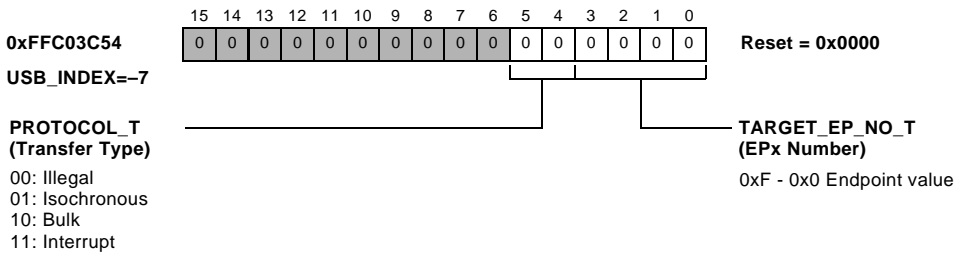


Figure 32-27. USB Tx Type (USB_TXTYPE) Register

USB NAK Limit 0 (USB_NAKLIMIT0) Register

The `USB_NAKLIMIT0` register (see [Figure 32-28](#)) determines the number of frames/micro-frames after which the endpoint should timeout on receiving a stream of NAK responses for bulk endpoints.

USB NAK Limit 0 Register (USB_NAKLIMIT0)

Read Only

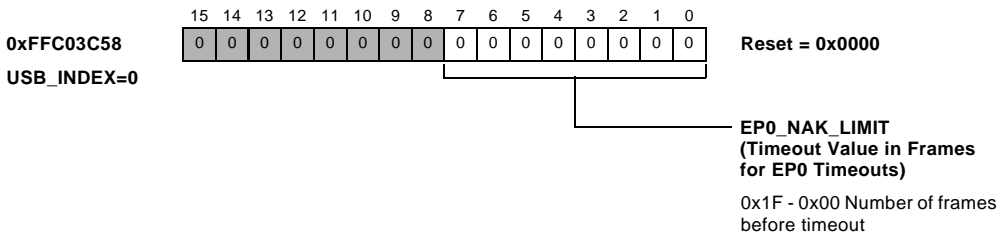


Figure 32-28. USB NAK Limit 0 (USB_NAKLIMIT0) Register

USB Tx Interval (USB_TXINTERVAL) Register

The USB_TXINTERVAL register (see [Figure 32-29](#)) 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

USB Tx Interval Register (USB_TXINTERVAL)

Read/Write

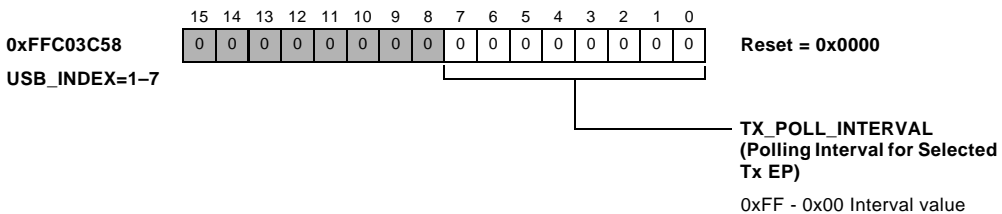


Figure 32-29. USB Tx Interval (USB_TXINTERVAL) Register

[Table 32-5](#) relates transfer types to TX_POLL_INTERVAL values (number of frames).

Table 32-4. 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 Rx Type (USB_RXTYPE) Register

The USB_RXTYPE register (see Figure 32-30) 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.

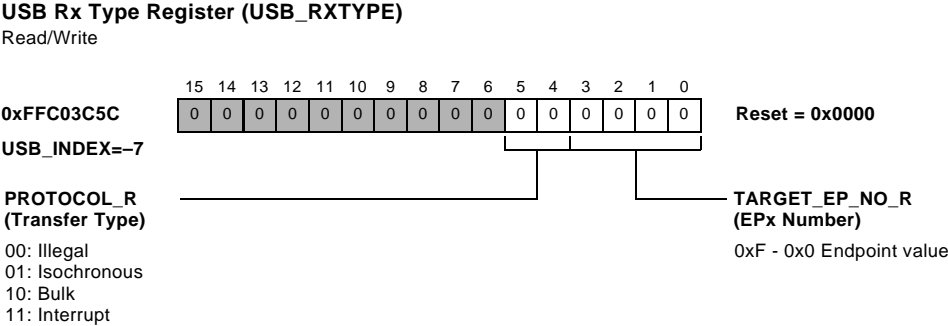


Figure 32-30. USB Tx Type (USB_TXTYPE) Register

USB Rx Interval (USB_RXINTERVAL) Register

The `USB_RXINTERVAL` register (see [Figure 32-31](#)) 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.

USB Rx Interval Register (USB_RXINTERVAL)

Read/Write

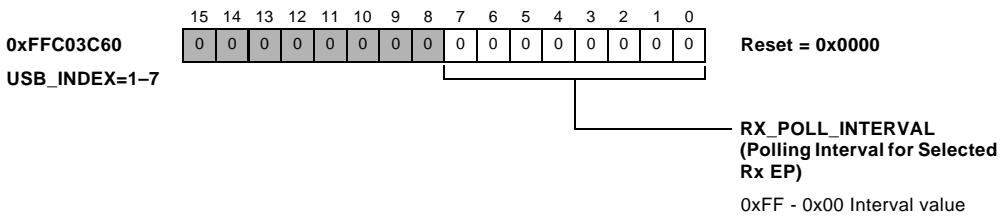


Figure 32-31. USB Rx Interval (USB_RXINTERVAL) Register

[Table 32-5](#) relates transfer types to `RX_POLL_INTERVAL` values (number of frames).

Table 32-5. 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 32-32](#)) selects the size in bytes of the packet/transfer which is about to be written into an Tx endpoint FIFO.

USB Tx Byte Count EPx Register (USB_TXCOUNT)

Read Only

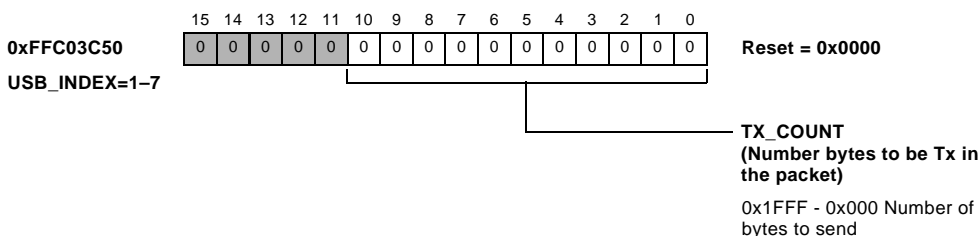


Figure 32-32. USB Tx Byte Count EPx (USB_TXCOUNT) Register

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 an Tx endpoint FIFO. The value is decremented by two when the processor core writes into the corresponding `USB_EPx_FIFO` high word address and is decremented by one when the processor core writes a byte into the FIFO using the corresponding `USB_EPx_FIFO` low word address. If the count itself reaches 0001h (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 32-56 for more details on `USB_TXCOUNT`’s usage.

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 32-55.

USB OTG Device Control (USB_OTG_DEV_CTL) Register

The USB_OTG_DEV_CTL register (see Figure 32-33) selects whether the USB controller is operating in peripheral mode or in host mode, and for controlling and monitoring the USB VBUS line.

USB OTG Device Control Register (USB_OTG_DEV_CTL)

Read/Write

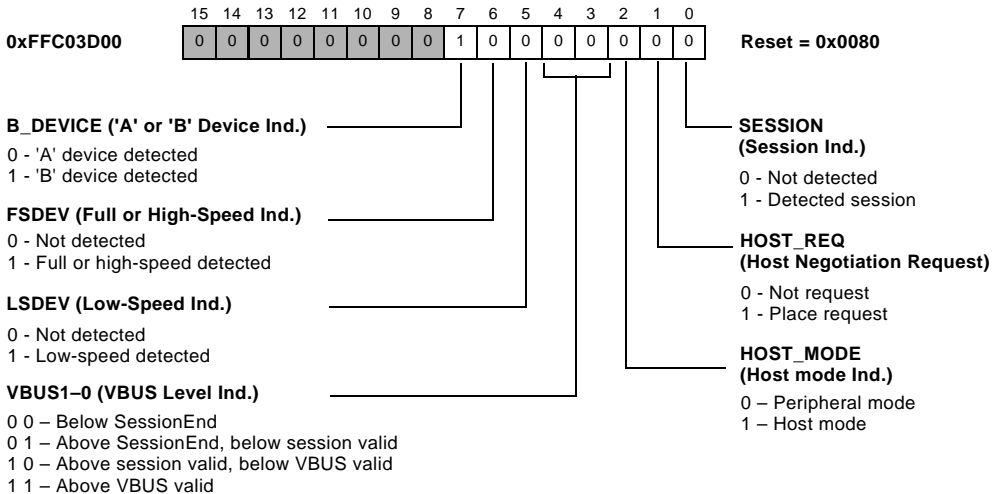


Figure 32-33. USB OTG Device Control (USB_OTG_DEV_CTL) Register

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.

USB OTG Registers

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 32-34](#)) 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.

USB OTG VBUS Interrupt Register (USB_OTG_VBUS_IRQ)

Read/Write

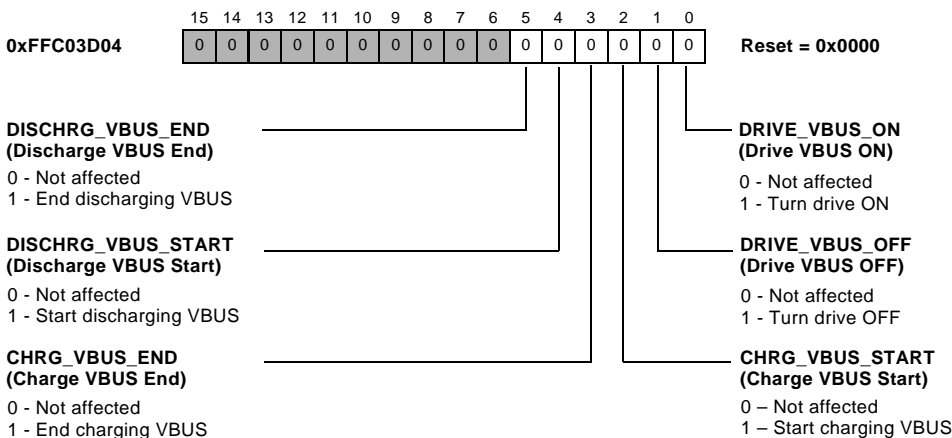


Figure 32-34. USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register

Because the charge pump and VBUS charge/discharge circuit is located in an external component/chip from 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 >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 external control circuit is to begin charging of VBUS in order to signal SRP ('B' device only).

CHRG_VBUS_END

When `CHRG_VBUS_END` (bit 3) is set, this status bit indicates external VBUS control to end charging of VBUS ('B' device only).

DISCHRG_VBUS_START

When `DISCHRG_VBUS_START` (bit 4) is set, this status bit indicates VBUS is to be discharged in order to speed up VBUS discharging below Session-End threshold ('B' device only).

DISCHRG_VBUS_END

When `DISCHRG_VBUS_END` (bit 5) is set, this status bit indicates VBUS control required 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 32-35](#)) provides interrupt enable bits for the interrupt sources in USB_OTG_VBUS_IRQ.

USB OTG VBUS Mask Register (USB_OTG_VBUS_MASK)

Read/Write

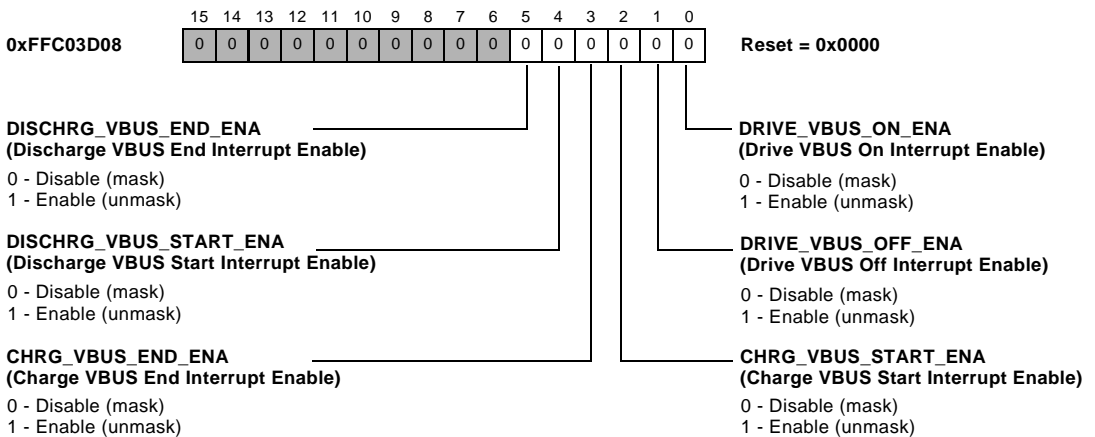


Figure 32-35. USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register

USB Link Info (USB_LINKINFO) Register

The USB_LINKINFO register (see [Figure 32-36](#)) allows some PHY-side delays to be specified.

USB Link Info Register (USB_LINKINFO)

Read/Write

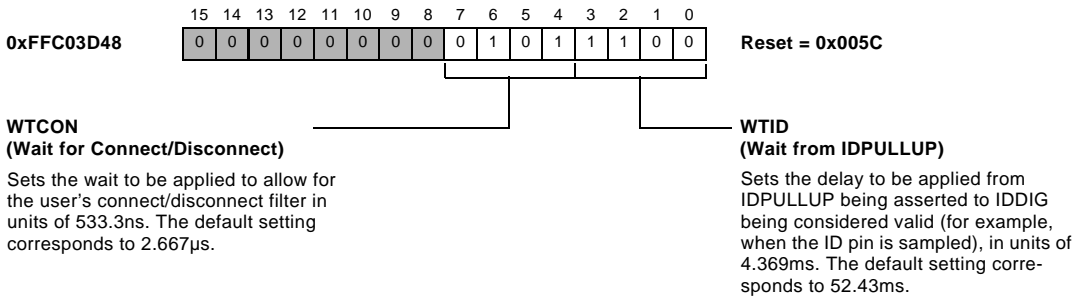


Figure 32-36. USB Link Info (USB_LINKINFO) Register

USB VBUS Pulse Length (USB_VPLEN) Register

The USB_VPLEN register (see [Figure 32-37](#)) defines the duration of the VBUS pulsing charge for SRP initiation.

USB VBUS Pulse Length Register (USB_VPLEN)

Read/Write

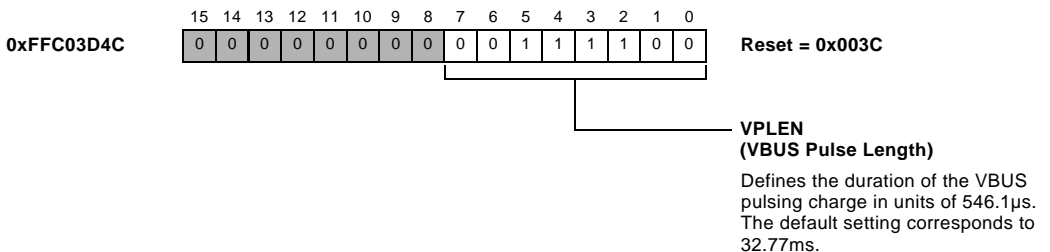


Figure 32-37. USB VBUS Pulse Length (USB_VPLEN) Register

USB High-Speed EOF 1 (USB_HS_EOF1) Register

The USB_HS_EOF1 register (see [Figure 32-38](#)) defines the minimum time gap that is to be allowed between the start of the last transaction and the EOF for high-speed transactions.

USB High-Speed EOF1 Register (USB_HS_EOF1)

Read/Write

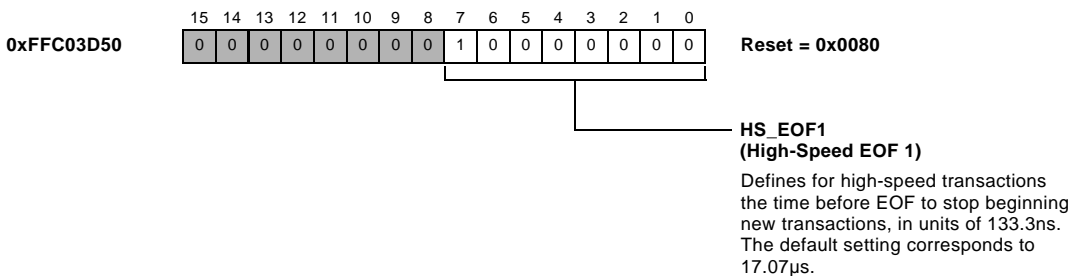


Figure 32-38. USB High-Speed EOF 1 (USB_HS_EOF1) Register

USB Full-Speed EOF 1 (USB_FS_EOF1) Register

The USB_FS_EOF1 register (see [Figure 32-39](#)) defines the minimum time gap that is to be allowed between the start of the last transaction and the EOF for full-speed transactions.

USB Full-Speed EOF1 Register (USB_FS_EOF1)

Read/Write

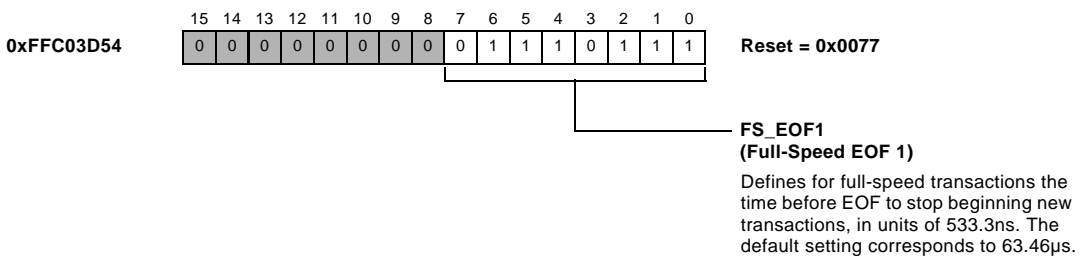


Figure 32-39. USB Full-Speed EOF 1 (USB_FS_EOF1) Register

USB Low-Speed EOF 1 (USB_LS_EOF1) Register

The USB_LS_EOF1 register (see [Figure 32-40](#)) defines the minimum time gap that is to be allowed between the start of the last transaction and the EOF for low-speed transactions.

USB Low-Speed EOF1 Register (USB_LS_EOF1)

Read/Write

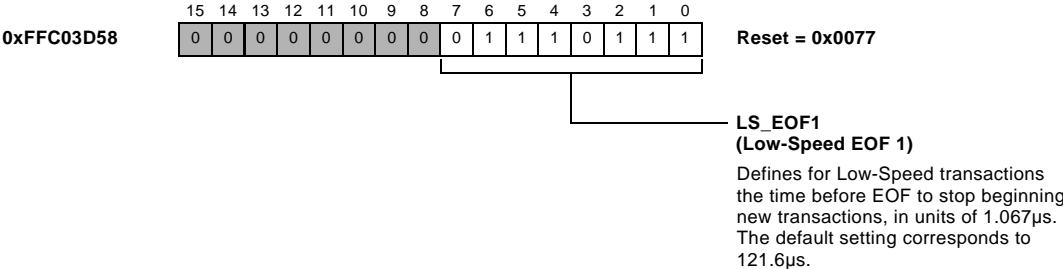


Figure 32-40. USB Low-Speed EOF 1 (USB_LS_EOF1) Register

USB APHY Control 2 (USB_APHY_CNTRL2) Register

The USB_APHY_CNTRL2 register (see [Figure 32-41](#)) helps to clean transition of the USB operation states from normal => suspend => hibernate => resume normal.

USB APHY Control 2 Register (USB_APHY_CNTRL2)

Read/Write

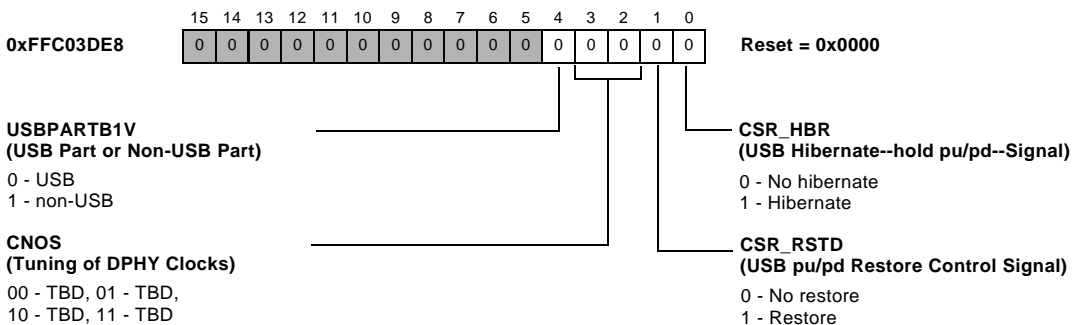


Figure 32-41. USB APHY Control 2 (USB_APHY_CNTRL2) Register

This register also is used tune the non-overlap between the clocks in the Digital PHY clock shaper circuit and it is also used to define whether the ADSP-BF54x processor is a USB part or non-USB part. The user is allowed to write to this register after the chip comes out of reset. This register is for Analog PHY and Digital PHY.

CSR_HBR

The CSR_HBR (bit 0) acts as the USB hibernate signal (hold pu/pd state) to signal the Analog PHY to hold the state of the pull-up and pull-downs on the D+ and D-.

CSR_RSTD

The CSR_RSTD (bit 1) acts as the USB pu/pd restore signal to signal the Analog PHY to release the holding state on the D+/D- pull-ups and pull-downs and give control to the USB controller to control the pull-ups and pull-downs.

CNOS[1:0]

The CNOS (bits 3–2) permits tuning non-overlap of the DPHY clocks.

USBPARTB1V

The USBPARTB1V (bit 4) defines whether the ADSP-BF54x processor is a USB part or non-USB part. This signal is used to switch off the PHY for power saving reasons.

USB PLL OSC Control (USB_PLLOSC_CTRL) Registers

The USB_PLLOSC_CTRL register (see [Figure 32-42](#)) program PLL and oscillator controls.

USB PLL OSC Control Register (USB_PLLOSC_CTRL)

Read/Write, Read-Only

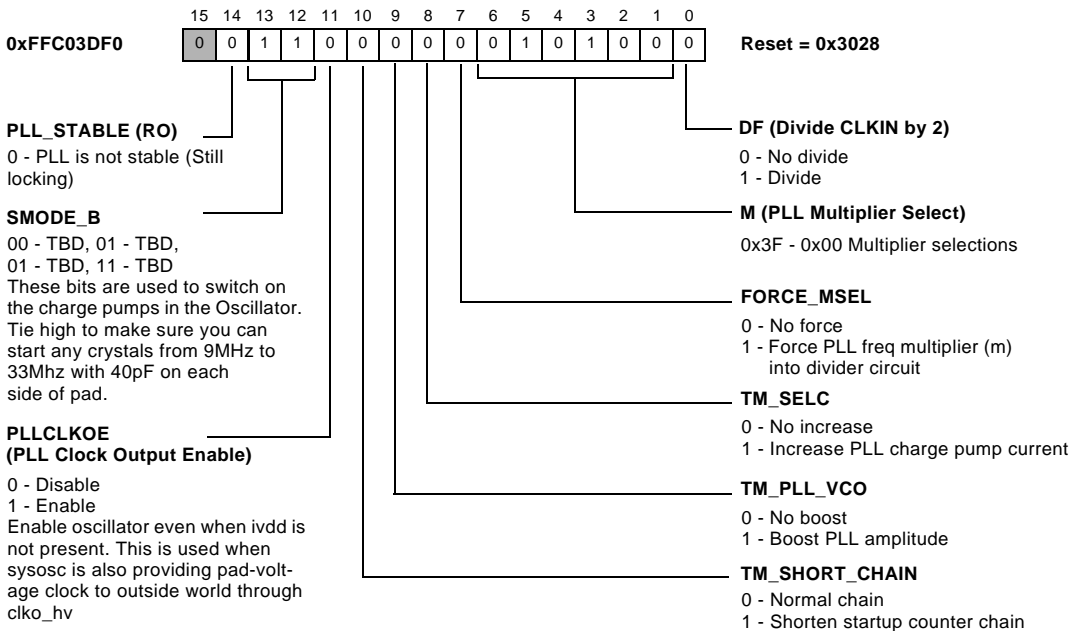


Figure 32-42. USB PLL OSC Control (USB_PLLOSC_CTRL) Register

USB SRP Clock Divider (USB_SRP_CLKDIV) Register

The USB_SRP_CLKDIV register (see [Figure 32-43](#)) programs the clock divider for sleep recovery of the USB peripheral (wakeup from sleep mode).

USB SRP Clock Divider Register (USB_SRP_CLKDIV)

Read/Write

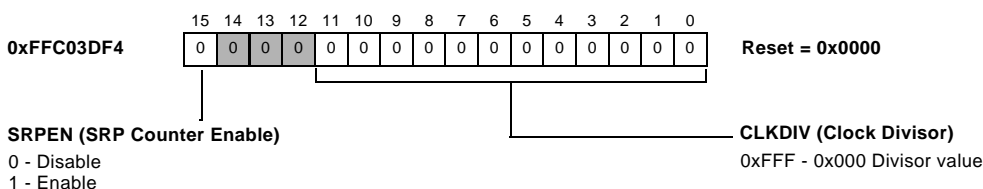


Figure 32-43. USB SRP Clock Divider (USB_SRP_CLKDIV) Register

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 is recommended to program a value in the divider register that would divide the peripheral clock frequency greater than or equal to 32KHz. Formula for calculating the value to be programmed into the USB_SRP_CLKDIV register:

$$\frac{\text{SCLK frequency in kHz}}{32} - 1 = \text{CLKDIV value}$$

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 32-44](#)) indicates which of the eight DMA master channels have a pending interrupt. The interrupt is generated when the corresponding DMA master channel's DMA Count register has reached zero. When the status is read by the processor core, the corresponding bit should have a 1 written to it by the software in order to clear the status.

USB DMA Interrupt Register (USB_DMA_INTERRUPT)

Read/Write

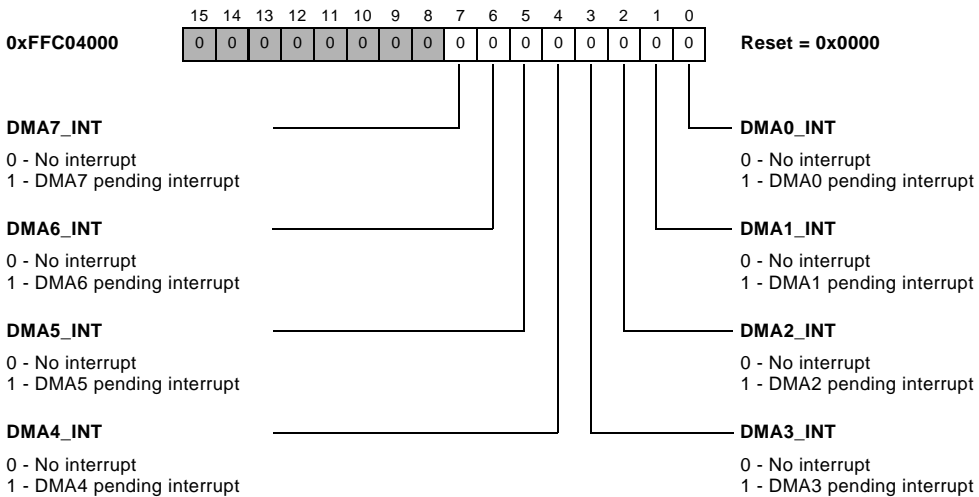


Figure 32-44. USB DMA Interrupt (USB_DMA_INTERRUPT) Register

USB DMAx Control (USB_DMAx_CONTROL) Registers

The `USB_DMAx_CONTROL` registers (see [Figure 32-42](#)) provide a DMA control register per 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 depending on the channel being used.

USB DMAx Control Registers (USB_DMAxCONTROL)

Read/Write

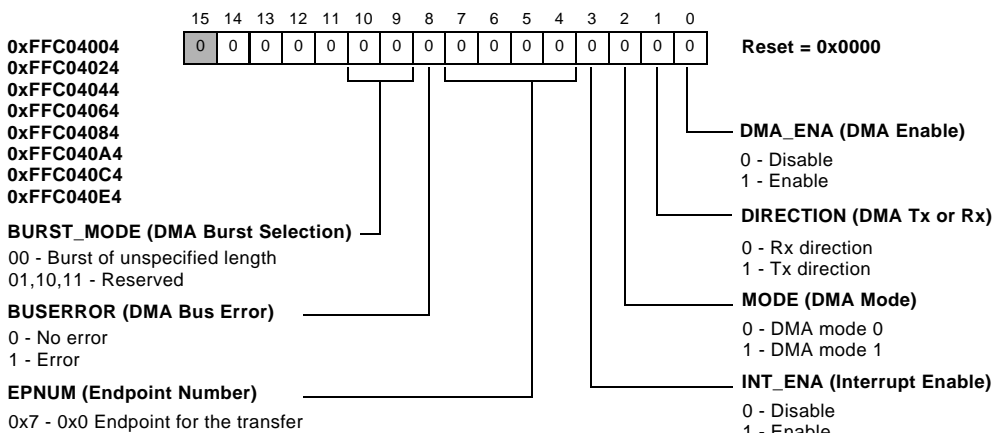


Figure 32-45. USB DMAx Control (USB_DMAxCONTROL) Registers

DMA_ENA

The `DMA_ENA` (bit 0) control bit enables the DMA master channel to allow it to begin transferring data between the FIFOs and on-chip memory.

DIRECTION

The `DIRECTION` (bit 1) control bit 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).

USB OTG Registers

MODE

The `MODE` (bit 2) control bit determines the DMA mode the channel is to operate in, DMA mode 0 or DMA mode 1.

INT_ENA

The `INT_ENA` (bit 3) control bit enables DMA interrupts (enable bit for the corresponding bit in the `USB_DMA_INTERRUPT` register).

EPNUM

The `EPNUM` (bits 7–4) control bits 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

The `BUSERROR` (bit 8) control bit indicates a peripheral bus error was encountered by the master channel.

BURST_MODE

The `BURST_MODE` (bits 10–9) control bits 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 32-47](#)) represent the least-significant half word of the full 32-bit DMA address, which indicates the location in on-chip memory where DMA data is written or read.

USB DMA Address Low Registers (USB_DMAxADDRLOW)

Read/Write

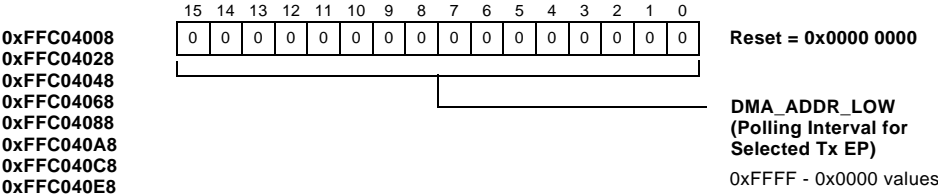


Figure 32-46. USB DMAx Address Low (USB_DMAxADDRLOW) Registers

USB DMAx Address High (USB_DMAxADDRHIGH) Registers

The USB_DMAxADDRHIGH registers (see [Figure 32-47](#)) represent the most-significant half word of the full 32-bit DMA address, which indicates the location in on-chip memory where DMA data is written or read.

USB DMA Address High Registers (USB_DMAxADDRHIGH)

Read/Write

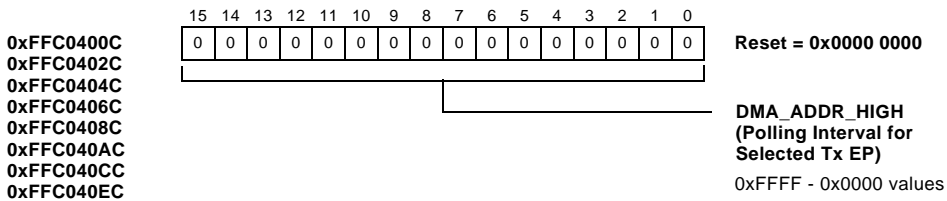


Figure 32-47. USB DMAx Address High (USB_DMAxADDRHIGH) Registers

USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers

The USB_DMAxCOUNTHIGH registers (see [Figure 32-49](#)) represent 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.

USB DMA Count High Registers (USB_DMAxCOUNTHIGH)

Read/Write

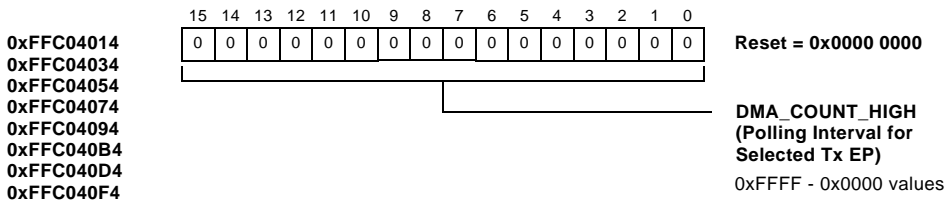


Figure 32-49. USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers

Programming Examples

The following programming examples (... through ...) describe the USB operations.

References

The following in are useful sources of reference information on 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

Device on USB with a mini-A plug inserted into its receptacle. The 'A' device always supplies power to VBUS.

'B' Device

Device on USB with a Standard-B or mini-B plug inserted into its receptacle. The 'B' device starts a session as the peripheral.

Bi-directional endpoint

Endpoint that appears to USB to be bi-directional (can concurrently support receive and transfer packets).

Glossary of USB Terms

Control endpoint

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 bi-directional endpoint 0.

Dual Role Device

USB device that can operate either as the USB host in an OTG session or as a traditional USB peripheral.

Endpoint

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

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

Lowest level of data exchange on USB, the size of which 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/down resistors as well as device ID and VBUS level detection.

Session

A period of time when USB transfers take place within an OTG connection, which can be initiated by the 'A' (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 transactions in sequence

Uni-directional 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

Glossary of USB Terms

A SYSTEM MMR ASSIGNMENTS

This appendix lists memory-mapped registers (MMR) addresses and register names for the system memory-mapped registers (MMRs), core timer registers, and processor-specific memory registers mentioned in this manual. Registers are listed in order by their memory-mapped address.

This chapter includes the following sections:

- [“Dynamic Power Management Registers” on page A-3](#)
- [“System Reset and Interrupt Control Registers” on page A-3](#)
- [“Watchdog Timer Registers” on page A-3](#)
- [“Real-Time Clock Registers” on page A-4](#)
- [“Ports Registers” on page A-4](#)
- [“Timer Registers” on page A-4](#)
- [“DMA/Memory DMA Control Registers” on page A-5](#)
- [“External Bus Interface Unit Registers” on page A-4](#)
- [“Handshake MDMA Control Registers” on page A-5](#)
- [“Host DMA Registers” on page A-5](#)
- [“PIXC Registers” on page A-5](#)
- [“Rotary Counter Registers” on page A-5](#)
- [“Security Registers” on page A-6](#)

- “Core Timer Registers” on page A-6
- “Processor-Specific Memory Registers” on page A-6
- “MXVR Registers” on page A-7
- “Keypad Registers” on page A-13
- “SDH Registers” on page A-13
- “ATAPI Registers” on page A-16
- “NAND Flash Controller Registers” on page A-18
- “EPPI1 Registers” on page A-19
- “EPPI2 Registers” on page A-20
- “CANx Registers” on page A-22
- “SPI0 Controller Registers” on page A-32
- “SPI1 Controller Registers” on page A-32
- “TWI Registers” on page A-33
- “SPORT0 Controller Registers” on page A-35
- “SPORT1 Controller Registers” on page A-37
- “UART0 Controller Registers” on page A-43
- “UART1 Controller Registers” on page A-44
- “UART2 Controller Registers” on page A-45
- “UART3 Controller Registers” on page A-46
- “USB OTG Registers” on page A-47

The following 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 the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

System Reset and Interrupt Control Registers

System reset and interrupt control registers (0xFFC0 0100 – 0xFFC0 01FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Real-Time Clock Registers

Real-time clock registers (0xFFC0 0300 – 0xFFC0 03FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Ports Registers

Ports registers (port F: 0xFFC0 0700 – 0xFFC0 07FF, port G: 0xFFC0 1500 – 0xFFC0 15FF, port H: 0xFFC0 1700 – 0xFFC0 17FF, pin control: 0xFFC0 3200 – 0xFFC0 32FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

External Bus Interface Unit Registers

External bus interface unit registers (0xFFC0 0A00 – 0xFFC0 0AFF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference*.

DMA/Memory DMA Control Registers

DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Handshake MDMA Control Registers

HMDMA registers (0xFFC0 3300 – 0xFFC0 33FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Host DMA Registers

Host DMA registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

PIXC Registers

Pixel compositor registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Rotary Counter Registers

Rotary Counter registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Security Registers

Security registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

MXVR Registers

Table A-1 shows the MXVR register memory map.

Table A-1. MXVR Memory Map

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 2700	MXVR_CONFIG	configuration register	16 R/W	0x1FCA
0xFFC0 2704	Reserved	–	–	–
0xFFC0 2708	MXVR_STATE_0	state register 0	32 RO	0x0000 0000
0xFFC0 270C	MXVR_STATE_1	state register 1	32 RO	0x0000 0000
0xFFC0 2710	MXVR_INT_STAT_0	interrupt status register 0	32 R/W	0x0000 0000
0xFFC0 2714	MXVR_INT_STAT_1	interrupt status register 1	32 R/W	0x0000 0000
0xFFC0 2718	MXVR_INT_EN_0	interrupt enable register 0	32 R/W	0x0000 0000
0xFFC0 271C	MXVR_INT_EN_1	interrupt enable register 1	32 R/W	0x0000 0000
0xFFC0 2720	MXVR_POSITION	node position register	16 RO	0x8000
0xFFC0 2724	MXVR_MAX_POSITION	maximum node position register	16 RO	0x0000
0xFFC0 2728	MXVR_DELAY	node frame delay register	16 RO	0x8000
0xFFC0 272C	MXVR_MAX_DELAY	maximum node frame delay register	16 RO	0x0000
0xFFC0 2730	MXVR_LADDR	logical address register	32 R/W	0x0000 0FFF
0xFFC0 2734	MXVR_GADDR	group address register	16 R/W	0x0000
0xFFC0 2738	MXVR_AADDR	alternate address register	32 R/W	0x0000 0FFF
0xFFC0 273C	MXVR_ALLOC_0	allocation table register 0	32 RO	0xFFFF XXXX
0xFFC0 2740	MXVR_ALLOC_1	allocation table register 1	32 RO	0xFFFF XXXX
0xFFC0 2744	MXVR_ALLOC_2	allocation table register 2	32 RO	0xFFFF XXXX
0xFFC0 2748	MXVR_ALLOC_3	allocation table register 3	32 RO	0xFFFF XXXX
0xFFC0 274C	MXVR_ALLOC_4	allocation table register 4	32 RO	0xFFFF XXXX
0xFFC0 2750	MXVR_ALLOC_5	allocation table register 5	32 RO	0xFFFF XXXX
0xFFC0 2754	MXVR_ALLOC_6	allocation table register 6	32 RO	0xFFFF XXXX
0xFFC0 2758	MXVR_ALLOC_7	allocation table register 7	32 RO	0xFFFF XXXX
0xFFC0 275C	MXVR_ALLOC_8	allocation table register 8	32 RO	0xFFFF XXXX

MXVR Registers

Table A-1. MXVR Memory Map (Cont'd)

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 2760	MXVR_ALLOC_9	allocation table register 9	32 RO	0XXXXX XXXX
0xFFC0 2764	MXVR_ALLOC_10	allocation table register 10	32 RO	0XXXXX XXXX
0xFFC0 2768	MXVR_ALLOC_11	allocation table register 11	32 RO	0XXXXX XXXX
0xFFC0 276C	MXVR_ALLOC_12	allocation table register 12	32 RO	0XXXXX XXXX
0xFFC0 2770	MXVR_ALLOC_13	allocation table register 13	32 RO	0XXXXX XXXX
0xFFC0 2774	MXVR_ALLOC_14	allocation table register 14	32 RO	0XXXXX XXXX
0xFFC0 2778	MXVR_SYNC_LCHAN_0	synchronous data logical channel assignment register 0	32 R/W	0xFFFF FFFF
0xFFC0 277C	MXVR_SYNC_LCHAN_1	synchronous data logical channel assignment register 1	32 R/W	0xFFFF FFFF
0xFFC0 2780	MXVR_SYNC_LCHAN_2	synchronous data logical channel assignment register 2	32 R/W	0xFFFF FFFF
0xFFC0 2784	MXVR_SYNC_LCHAN_3	synchronous data logical channel assignment register 3	32 R/W	0xFFFF FFFF
0xFFC0 2788	MXVR_SYNC_LCHAN_4	synchronous data logical channel assignment register 4	32 R/W	0xFFFF FFFF
0xFFC0 278C	MXVR_SYNC_LCHAN_5	synchronous data logical channel assignment register 5	32 R/W	0xFFFF FFFF
0xFFC0 2790	MXVR_SYNC_LCHAN_6	synchronous data logical channel assignment register 6	32 R/W	0xFFFF FFFF
0xFFC0 2794	MXVR_SYNC_LCHAN_7	synchronous data logical channel assignment register 7	32 R/W	0xFFFF FFFF
0xFFC0 2798	MXVR_DMA0_CONFIG	synchronous data DMA0 configuration register	32 R/W	0x0000 0000
0xFFC0 279C	MXVR_DMA0_START_ADDR	synchronous data DMA0 start address register	32 R/W	0xFF00 0000
0xFFC0 27A0	MXVR_DMA0_COUNT	synchronous data DMA0 loop count register	16 R/W	0x0001
0xFFC0 27A4	MXVR_DMA0_CURR_ADDR	synchronous data DMA0 current address register	32 RO	0xFF00 0000
0xFFC0 27A8	MXVR_DMA0_CURR_COUNT	synchronous data DMA0 current loop count register	16 RO	0x0000

Table A-1. MXVR Memory Map (Cont'd)

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 27AC	MXVR_DMA1_CONFIG	synchronous data DMA1 configuration register	32 R/W	0x0000 0000
0xFFC0 27B0	MXVR_DMA1_START_ADDR	synchronous data DMA1 start address register	32 R/W	0xFF00 0000
0xFFC0 27B4	MXVR_DMA1_COUNT	synchronous data DMA1 loop count register	16 R/W	0x0001
0xFFC0 27B8	MXVR_DMA1_CURR_ADDR	synchronous data DMA1 current address register	32 RO	0xFF00 0000
0xFFC0 27BC	MXVR_DMA1_CURR_COUNT	synchronous data DMA1 current Loop count register	16 RO	0x0000
0xFFC0 27C0	MXVR_DMA2_CONFIG	synchronous data DMA2 configuration register	32 R/W	0x0000 0000
0xFFC0 27C4	MXVR_DMA2_START_ADDR	synchronous data DMA2 start address register	32 R/W	0xFF00 0000
0xFFC0 27C8	MXVR_DMA2_COUNT	synchronous data DMA2 loop count register	16 R/W	0x0001
0xFFC0 27CC	MXVR_DMA2_CURR_ADDR	synchronous data DMA2 current address register	32 RO	0xFF00 0000
0xFFC0 27D0	MXVR_DMA2_CURR_COUNT	synchronous data DMA2 current loop count register	16 RO	0x0000
0xFFC0 27D4	MXVR_DMA3_CONFIG	synchronous data DMA3 configuration register	32 R/W	0x0000 0000
0xFFC0 27D8	MXVR_DMA3_START_ADDR	synchronous data DMA3 start address register	32 R/W	0xFF00 0000
0xFFC0 27DC	MXVR_DMA3_COUNT	synchronous data DMA3 loop count register	16 R/W	0x0001
0xFFC0 27E0	MXVR_DMA3_CURR_ADDR	synchronous data DMA3 current address register	32 RO	0xFF00 0000
0xFFC0 27E4	MXVR_DMA3_CURR_COUNT	synchronous data DMA3 current loop count register	16 RO	0x0000
0xFFC0 27E8	MXVR_DMA4_CONFIG	synchronous data DMA4 configuration register	32 R/W	0x0000 0000

MXVR Registers

Table A-1. MXVR Memory Map (Cont'd)

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 27EC	MXVR_DMA4_START_ADDR	synchronous data DMA4 start address register	32 R/W	0xFF00 0000
0xFFC0 27F0	MXVR_DMA4_COUNT	synchronous data DMA4 loop count register	16 R/W	0x0001
0xFFC0 27F4	MXVR_DMA4_CURR_ADDR	synchronous data DMA4 current address register	32 RO	0xFF00 0000
0xFFC0 27F8	MXVR_DMA4_CURR_COUNT	synchronous data DMA4 current loop count register	16 RO	0x0000
0xFFC0 27FC	MXVR_DMA5_CONFIG	synchronous data DMA5 configuration register	32 R/W	0x0000 0000
0xFFC0 2800	MXVR_DMA5_START_ADDR	synchronous data DMA5 start address register	32 R/W	0xFF00 0000
0xFFC0 2804	MXVR_DMA5_COUNT	synchronous data DMA5 loop count register	16 R/W	0x0000
0xFFC0 2808	MXVR_DMA5_CURR_ADDR	synchronous data DMA5 current address register	32 RO	0xFF00 0000
0xFFC0 280C	MXVR_DMA5_CURR_COUNT	synchronous data DMA5 current loop count register	16 RO	0x0000
0xFFC0 2810	MXVR_DMA6_CONFIG	synchronous data DMA6 configuration register	32 R/W	0x0000 0000
0xFFC0 2814	MXVR_DMA6_START_ADDR	synchronous data DMA6 start address register	32 R/W	0xFF00 0000
0xFFC0 2818	MXVR_DMA6_COUNT	synchronous data DMA6 loop count register	16 R/W	0x0001
0xFFC0 281C	MXVR_DMA6_CURR_ADDR	synchronous data DMA6 current address register	32 RO	0xFF00 0000
0xFFC0 2820	MXVR_DMA6_CURR_COUNT	synchronous data DMA6 current loop count register	16 RO	0x0000
0xFFC0 2824	MXVR_DMA7_CONFIG	synchronous data DMA7 configuration register	32 R/W	0x0000 0000
0xFFC0 2828	MXVR_DMA7_START_ADDR	synchronous data DMA7 start address register	32 R/W	0xFF00 0000

Table A-1. MXVR Memory Map (Cont'd)

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 282C	MXVR_DMA7_COUNT	synchronous data DMA7 loop count register	16 R/W	0x0001
0xFFC0 2830	MXVR_DMA7_CURR_ADDR	synchronous data DMA7 current address register	32 RO	0xFF00 0000
0xFFC0 2834	MXVR_DMA7_CURR_COUNT	synchronous data DMA7 current loop count register	16 RO	0x0000
0xFFC0 2838	MXVR_AP_CTL	asynchronous packet control register	16 R/W	0x0000
0xFFC0 283C	MXVR_APRB_START_ADDR	asynchronous packet receive buffer start address register	32 R/W	0xFF00 0000
0xFFC0 2840	MXVR_APRB_CURR_ADDR	asynchronous packet receive buffer current address register	32 RO	0xFF00 0000
0xFFC0 2844	MXVR_APTB_START_ADDR	asynchronous packet transmit buffer start address register	32 R/W	0xFF00 0000
0xFFC0 2848	MXVR_APTB_CURR_ADDR	asynchronous packet transmit buffer current address register	32 RO	0xFF00 0000
0xFFC0 284C	MXVR_CM_CTL	control message control register	32 R/W	0x0000 0000
0xFFC0 2850	MXVR_CMRB_START_ADDR	control message receive buffer start address register	32 R/W	0xFF00 0000
0xFFC0 2854	MXVR_CMRB_CURR_ADDR	control message receive buffer current address register	32 RO	0xFF00 0000
0xFFC0 2858	MXVR_CMTB_START_ADDR	control message transmit buffer start address register	32 R/W	0xFF00 0000
0xFFC0 285C	MXVR_CMTB_CURR_ADDR	control message transmit buffer current address register	32 RO	0xFF00 0000
0xFFC0 2860	MXVR_RRDB_START_ADDR	remote read buffer start address register	32 R/W	0xFF00 0000
0xFFC0 2864	MXVR_RRDB_CURR_ADDR	remote read buffer current address register	32 RO	0xFF00 0000
0xFFC0 2868	MXVR_PAT_DATA_0	pattern data register 0	32 R/W	0x0000 0000
0xFFC0 286C	MXVR_PAT_EN_0	pattern enable register 0	32 R/W	0x0000 0000
0xFFC0 2870	MXVR_PAT_DATA_1	pattern data register 1	32 R/W	0x0000 0000
0xFFC0 2874	MXVR_PAT_EN_1	pattern enable register 1	32 R/W	0x0000 0000

MXVR Registers

Table A-1. MXVR Memory Map (Cont'd)

Register Address	Register Name	Register Description	Size (bits)	Reset Value
0xFFC0 2878	MXVR_FRAME_CNT_0	frame counter 0	16 R/W	0x0000
0xFFC0 287C	MXVR_FRAME_CNT_1	frame counter 1	16 R/W	0x0000
0xFFC0 2880	MXVR_Routing_0	routing register 0	32 WO	0xFFFF XXXX
0xFFC0 2884	MXVR_Routing_1	routing register 1	32 WO	0xFFFF XXXX
0xFFC0 2888	MXVR_Routing_2	routing register 2	32 WO	0xFFFF XXXX
0xFFC0 288C	MXVR_Routing_3	routing register 3	32 WO	0xFFFF XXXX
0xFFC0 2890	MXVR_Routing_4	routing register 4	32 WO	0xFFFF XXXX
0xFFC0 2894	MXVR_Routing_5	routing register 5	32 WO	0xFFFF XXXX
0xFFC0 2898	MXVR_Routing_6	routing register 6	32 WO	0xFFFF XXXX
0xFFC0 289C	MXVR_Routing_7	routing register 7	32 WO	0xFFFF XXXX
0xFFC0 28A0	MXVR_Routing_8	routing register 8	32 WO	0xFFFF XXXX
0xFFC0 28A4	MXVR_Routing_9	routing register 9	32 WO	0xFFFF XXXX
0xFFC0 28A8	MXVR_Routing_10	routing register 10	32 WO	0xFFFF XXXX
0xFFC0 28AC	MXVR_Routing_11	routing register 11	32 WO	0xFFFF XXXX
0xFFC0 28B0	MXVR_Routing_12	routing register 12	32 WO	0xFFFF XXXX
0xFFC0 28B4	MXVR_Routing_13	routing register 13	32 WO	0xFFFF XXXX
0xFFC0 28B8	MXVR_Routing_14	routing register 14	32 WO	0xFFFF XXXX
0xFFC0 28BC	Reserved	–	–	–
0xFFC0 28C0	MXVR_BLOCK_CNT	block counter	16 R/W	0x0000
0xFFC0 28C4 to 0xFFC0 28CC	Reserved	–	–	–
0xFFC0 28D0	MXVR_CLK_CTL	clock control register	32 R/W	0x0202 0003
0xFFC0 28D4	MXVR_CDRPLL_CTL	clock/data recovery PLL ctrl register	32 R/W	0x0502 0820
0xFFC0 28D8	MXVR_FMPDLL_CTL	frequency mult. PLL ctrl register	32 R/W	0x1900 1020
0xFFC0 28DC	MXVR_PIN_CTL	pin control register	16 R/W	0x0000
0xFFC0 28E0	MXVR_SCLK_CNT	system clock counter register	16 R/W	0x0000
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-2](#).

Table A-2. Control/Status/Data Registers

Memory-mapped Address	Name	Description
0xFFC04100	KPAD_CTL	Keypad control register on page 22-11
0xFFC04104	KPAD_PRESCALE	Keypad prescale register on page 22-14
0xFFC04108	KPAD_MSEL	Keypad multiplier select register on page 22-16
0xFFC0410c	KPAD_ROWCOL	Keypad row-column register on page 22-16
0xFFC04110	KPAD_STAT	Keypad status register on page 22-20
0xFFC04114	KPAD_SOFTEVAL	Keypad software evaluate register on page 22-21

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.

SDH Registers

The SDH memory-mapped registers start at base address 0xFFC03900. In [Table A-3](#), 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-3. SDH Functional Registers

Memory-mapped Address	Register Name	Description
0xFFC03900	SDH_PWR_CTL	SDH power control register on page 23-22
0xFFC03904	SDH_CLK_CTL	SDH clock control register on page 23-23
0xFFC03908	SDH_ARGUMENT	SDH argument register on page 23-19
0xFFC0390c	SDH_COMMAND	SDH command register on page 23-24
0xFFC03910	SDH_RESP_CMD	SDH response command register on page 23-25
0xFFC03914	SDH_RESPONSE0	SDH response 0 register on page 23-26
0xFFC03918	SDH_RESPONSE1	SDH response 1 register on page 23-25
0xFFC0391c	SDH_RESPONSE2	SDH response 2 register on page 23-25
0xFFC03920	SDH_RESPONSE3	SDH response 3 register on page 23-25
0xFFC03924	SDH_DATA_TIMER	SDH data timer register on page 23-27
0xFFC03928	SDH_DATA_LGTH	SDH data length register on page 23-27
0xFFC0392c	SDH_DATA_CTL	SDH data control register on page 23-27
0xFFC03930	SDH_DATA_CNT	SDH data counter register on page 23-29

Table A-3. SDH Functional Registers (Cont'd)

Memory-mapped Address	Register Name	Description
0xFFC03934	SDH_STATUS	SDH status register on page 23-30
0xFFC03938	SDH_STATUS_CLR	SDH status clear register on page 23-32
0xFFC0393c	SDH_MASK0	SDH interrupt 0 mask register on page 23-33
0xFFC03940	SDH_MASK1	SDH interrupt 1 mask register on page 23-33
0xFFC03944	Reserved	–
0xFFC03948	SDH_FIFO_CNT	SDH FIFO counter register on page 23-34
0xFFC0394c ... 0xFFC0397c	Reserved	–
0xFFC03980	SDH_FIFOx	SDH data FIFO registers on page 23-34
0xFFC03984 ... 0xFFC03988	Reserved	–
0xFFC039c0	SDH_E_STATUS	SDH exception status register on page 23-35
0xFFC039c4	SDH_E_MASK	SDH exception mask register on page 23-36
0xFFC039c8	SDH_CFG	SDH configuration register on page 23-37
0xFFC039cc	SDH_RD_WAIT_EN	SDH read wait enable register on page 23-38
0xFFC039d0 ... 0xFFC039ec	SDH_PIDx	SDH peripheral identification registers (8-bit values) on page 23-38

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-4](#) lists the ATAPI memory-mapped registers, starting at base address 0xFFC03800. Register addresses are given relative to the base address.

Table A-4. ATAPI Core Registers

Memory-mapped Address	Register Name	Description
0xFFC03800	ATAPI_CONTROL	ATAPI control on page 24-50
0xFFC03804	ATAPI_STATUS	ATAPI status on page 24-52
0xFFC03808	ATAPI_DEV_ADDR	ATAPI device address on page 24-53
0xFFC0380C	ATAPI_DEV_TXBUF	ATAPI device transmit buffer on page 24-54
0xFFC03810	ATAPI_DEV_RXBUF	ATAPI device receive buffer on page 24-55
0xFFC03814	ATAPI_INT_MASK	ATAPI interrupt mask on page 24-56
0xFFC03818	ATAPI_INT_STATUS	ATAPI interrupt status on page 24-57
0xFFC0381C	ATAPI_XFER_LEN	ATAPI transfer length on page 24-59
0xFFC03820	ATAPI_LINE_STATUS	ATAPI line status on page 24-60
0xFFC03824	ATAPI_SM_STATE	ATAPI state machine status on page 24-61
0xFFC03828	ATAPI_TERMINATE	ATAPI terminate on page 24-61
0xFFC0382C	ATAPI_PIO_TFRcnt	ATAPI PIO transfer count on page 24-62
0xFFC03830	ATAPI_DMA_TFRcnt	ATAPI multi-word DMA transfer count on page 24-63
0xFFC03834	ATAPI_ULTRA_IN_TFRcnt	ATAPI ultra-DMA in transfer count on page 24-63

Table A-4. ATAPI Core Registers (Cont'd)

Memory-mapped Address	Register Name	Description
0xFFC03838	ATAPI_ULTRA_OUT_TF RCNT	ATAPI ultra-DMA out transfer count on page 24-64
0xFFC03840	ATAPI_REG_TIM_0	ATAPI register transfer timing 0 on page 24-64
0xFFC03844	ATAPI_PIO_TIM_0	ATAPI programmed I/O timing 0 on page 24-65
0xFFC03848	ATAPI_PIO_TIM_1	ATAPI programmed I/O timing 1 on page 24-65
0xFFC03850	ATAPI_MULTI_TIM_0	ATAPI multi-DMA timing 0 on page 24-66
0xFFC03854	ATAPI_MULTI_TIM_1	ATAPI multi-DMA timing 1 on page 24-66
0xFFC03858	ATAPI_MULTI_TIM_2	ATAPI multi-DMA timing 2 on page 24-67
0xFFC03860	ATAPI_ULTRA_TIM_0	ATAPI ultra-DMA timing 0 on page 24-67
0xFFC03864	ATAPI_ULTRA_TIM_1	ATAPI ultra-DMA timing 1 on page 24-67
0xFFC03868	ATAPI_ULTRA_TIM_2	ATAPI ultra-DMA timing 2 on page 24-67
0xFFC0386C	ATAPI_ULTRA_TIM_3	ATAPI ultra-DMA timing 3 on page 24-67

NAND Flash Controller Registers

Table A-6 lists all of the NFC memory-mapped registers.

Table A-5. NFC Memory-Mapped Registers

Memory-mapped Address	Register Name	Description
0xFFC0 3B00	NFC_CTL	NFC control register on page 25-20
0xFFC0 3B04	NFC_STAT	NFC status register on page 25-21
0xFFC0 3B08	NFC_IRQSTAT	NFC interrupt status register on page 25-22
0xFFC0 3B0C	NFC_IRQMASK	NFC interrupt mask register on page 25-23
0xFFC0 3B10	NFC_ECC0	NFC ECC register 0 on page 25-24
0xFFC0 3B14	NFC_ECC1	NFC ECC register 1 on page 25-24
0xFFC0 3B18	NFC_ECC2	NFC ECC register 2 on page 25-24
0xFFC0 3B1C	NFC_ECC3	NCF ECC register 3 on page 25-24
0xFFC0 3B20	NFC_COUNT	NFC count register on page 25-25
0xFFC0 3B24	NFC_RST	NFC reset register on page 25-25
0xFFC0 3B28	NFC_PGCTL	NFC page control register on page 25-26
0xFFC0 3B2C	NFC_READ	NFC read data register on page 25-27

Table A-5. NFC Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Register Name	Description
0xFFC0 3B40	NFC_ADDR	NFC address register on page 25-27
0xFFC0 3B44	NFC_CMD	NFC command register on page 25-28
0xFFC0 3B48	NFC_DATA_WR	NFC data write register on page 25-29
0xFFC0 3B4C	NFC_DATA_RD	NFC data read register on page 25-29

EPPI1 Registers

PPI1 registers are listed in [Table A-7](#).

Table A-6. PPI1 Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 1300	PPI1_STATUS	“EPPI Status Register” on page 26-90
0xFFC0 1304	PPI1_HCOUNT	“EPPI Horizontal Transfer Count Register” on page 26-95
0xFFC0 1308	PPI1_HDELAY	“EPPI Horizontal Delay Register” on page 26-94
0xFFC0 130C	PPI1_VCOUNT	“EPPI Vertical Transfer Count Register” on page 26-94
0xFFC0 1310	PPI1_VDELAY	“EPPI Vertical Delay Count Register” on page 26-93
0xFFC0 1314	PPI1_FRAME	“EPPI Lines per Frame Register” on page 26-92

EPPI2 Registers

Table A-6. PPI1 Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 1318	PPI1_LINE	“EPPI Samples per Line Register” on page 26-93
0xFFC0 131C	PPI1_CLKDIV	“EPPI Clock Divide Register (PPIx_CLK)” on page 26-96
0xFFC0 1320	PPI1_CONTROL	“PPIx Control Register, Upper Half” on page 26-83
0xFFC0 1324	PPI1_FSIW_HBL	“EPPI FS1 Width / Horizontal Blanking Samples per Line Register” on page 26-97
0xFFC0 1328	PPI1_FSIP_AVPL	“EPPI FS1 Period Register / EPPI Active Video Samples per Line Register (PPIx_FSIP_AVPL)” on page 26-99
0xFFC0 132C	PPI1_FS2W_LVB	“EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register” on page 26-98
0xFFC0 1330	PPI1_FS2P_LAVF	“EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (PPIx_FS2_LAVF)” on page 26-100
0xFFC0 1334	PPI1_CLIP	“EPPI Clipping Register (PPIx_CLIP)” on page 26-101

EPPI2 Registers

PPI2 registers are listed in [Table A-7](#).

Table A-7. PPI2 Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2900	PPI2_STATUS	“EPPI Status Register” on page 26-90
0xFFC0 2904	PPI2_HCOUNT	“EPPI Horizontal Transfer Count Register” on page 26-95

Table A-7. PPI2 Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 2908	PPI2_HDELAY	“EPPI Horizontal Delay Register” on page 26-94
0xFFC0 290C	PPI2_VCOUNT	“EPPI Vertical Transfer Count Register” on page 26-94
0xFFC0 2910	PPI2_VDELAY	“EPPI Vertical Delay Count Register” on page 26-93
0xFFC0 2914	PPI2_FRAME	“EPPI Lines per Frame Register” on page 26-92
0xFFC0 2918	PPI2_LINE	“EPPI Samples per Line Register” on page 26-93
0xFFC0 291C	PPI2_CLKDIV	“EPPI Clock Divide Register (PPIx_CLK)” on page 26-96
0xFFC0 2920	PPI2_CONTROL	“PPIx Control Register, Upper Half” on page 26-83
0xFFC0 2924	PPI2_FSIW_HBL	“EPPI FS1 Width / Horizontal Blanking Samples per Line Register” on page 26-97
0xFFC0 2928	PPI2_FSIP_AVPL	“EPPI FS1 Period Register / EPPI Active Video Samples per Line Register (PPIx_FSIP_AVPL)” on page 26-99
0xFFC0 292C	PPI2_FS2W_LVB	“EPPI FS2 Width Register/EPPI Lines of Vertical Blanking Register” on page 26-98
0xFFC0 2930	PPI2_FS2P_LAVF	“EPPI FS2 Period Register/EPPI Lines of Active Video per Frame Register (PPIx_FS2_LAVF)” on page 26-100
0xFFC0 2934	PPI2_CLIP	“EPPI Clipping Register (PPIx_CLIP)” on page 26-101

CANx Registers

CANx registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in [Table A-8](#) through [Table A-11](#).

Table A-8. CANx Control and Configuration Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2A00	CANx_MC1	“Mailbox Configuration Register 1” on page 27-73
0xFFC0 2A04	CANx_MD1	“Mailbox Direction Register 1” on page 27-74
0xFFC0 2A08	CANx_TRS1	“Transmission Request Set Register 1” on page 27-78
0xFFC0 2A0C	CANx_TRR1	“Transmission Request Reset Register 1” on page 27-79
0xFFC0 2A10	CANx_TA1	“Transmission Acknowledge Register 1” on page 27-81
0xFFC0 2A14	CANx_AA1	“Abort Acknowledge Register 1” on page 27-80
0xFFC0 2A18	CANx_RMP1	“Receive Message Pending Register 1” on page 27-75
0xFFC0 2A1C	CANx_RML1	“Receive Message Lost Register 1” on page 27-76
0xFFC0 2A20	CANx_MBTIF1	“Mailbox Transmit Interrupt Flag Register 1” on page 27-85
0xFFC0 2A24	CANx_MBRIF1	“Mailbox Receive Interrupt Flag Register 1” on page 27-86
0xFFC0 2A28	CANx_MBIM1	“Mailbox Interrupt Mask Register 1” on page 27-84
0xFFC0 2A2C	CANx_RFH1	“Remote Frame Handling Register 1” on page 27-83

Table A-8. CANx Control and Configuration Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 2A30	CANx_OPSS1	“Overwrite Protection/Single Shot Transmission Register 1” on page 27-77
0xFFC0 2A40	CANx_MC2	“Mailbox Configuration Register 2” on page 27-73
0xFFC0 2A44	CANx_MD2	“Mailbox Direction Register 2” on page 27-74
0xFFC0 2A48	CANx_TRS2	“Transmission Request Set Register 2” on page 27-78
0xFFC0 2A4C	CANx_TRR2	“Transmission Request Reset Register 2” on page 27-79
0xFFC0 2A50	CANx_TA2	“Transmission Acknowledge Register 2” on page 27-81
0xFFC0 2A54	CANx_AA2	“Abort Acknowledge Register 2” on page 27-80
0xFFC0 2A58	CANx_RMP2	“Receive Message Pending Register 2” on page 27-75
0xFFC0 2A5C	CANx_RML2	“Receive Message Lost Register 2” on page 27-76
0xFFC0 2A60	CANx_MBTIF2	“Mailbox Transmit Interrupt Flag Register 2” on page 27-85
0xFFC0 2A64	CANx_MBRIF2	“Mailbox Receive Interrupt Flag Register 2” on page 27-86
0xFFC0 2A68	CANx_MBIM2	“Mailbox Interrupt Mask Register 2” on page 27-84
0xFFC0 2A6C	CANx_RFH2	“Remote Frame Handling Register 2” on page 27-83
0xFFC0 2A70	CANx_OPSS2	“Overwrite Protection/Single Shot Transmission Register 2” on page 27-77
0xFFC0 2A80	CANx_CLOCK	“CAN Clock Registers” on page 27-48
0xFFC0 2A84	CANx_TIMING	“CAN Timing Registers” on page 27-49

CANx Registers

Table A-8. CANx Control and Configuration Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0_2A88	CANx_DEBUG	“CAN Timing Registers” on page 27-49
0xFFC0_2A8C	CANx_STATUS	“Global CAN Status Registers” on page 27-47
0xFFC0_2A90	CANx_CEC	“Error Counter Register” on page 27-89
0xFFC0_2A94	CANx_GIS	“Global CAN Interrupt Status Registers” on page 27-51
0xFFC0_2A98	CANx_GIM	“Global CAN Interrupt Mask Registers” on page 27-50
0xFFC0_2A9C	CANx_GIF	“Global CAN Interrupt Flag Registers” on page 27-52
0xFFC0_2AA0	CANx_CONTROL	“Master Control Registers” on page 27-46
0xFFC0_2AA4	CANx_INTR	“CAN Interrupt Registers” on page 27-49
0xFFC0_2AAC	CANx_MBTD	“Temporary Mailbox Disable Register” on page 27-82
0xFFC0_2AB0	CANx_EWR	“Error Counter Warning Level Register” on page 27-90
0xFFC0_2AB4	CANx_ESR	“Error Status Register” on page 27-89
0xFFC0_2AC4	CANx_UCCNT	“Universal Counter Register” on page 27-88
0xFFC0_2AC8	CANx_UCRC	“Universal Counter Reload/Capture Register” on page 27-88
0xFFC0_2ACC	CANx_UCCNF	“Universal Counter Configuration Mode Register” on page 27-87

Table A-9. CANx Mailbox Acceptance Mask Registers

Memory-mapped Address	Register Name	See Section
0xFFC0 2B00	CANx_AM00L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B04	CANx_AM00H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B08	CANx_AM01L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B0C	CANx_AM01H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B10	CANx_AM02L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B14	CANx_AM02H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B18	CANx_AM03L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B1C	CANx_AM03H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B20	CANx_AM04L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B24	CANx_AM04H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B28	CANx_AM05L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B2C	CANx_AM05H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B30	CANx_AM06L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2B34	CANx_AM06H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2B38	CANx_AM07L	“Acceptance Mask Registers (L)” on page 27-55

CANx Registers

Table A-9. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2B3C	CANx_AM07H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B40	CANx_AM08L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B44	CANx_AM08H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B48	CANx_AM09L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B4C	CANx_AM09H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B50	CANx_AM10L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B54	CANx_AM10H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B58	CANx_AM11L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B5C	CANx_AM11H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B60	CANx_AM12L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B64	CANx_AM12H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B68	CANx_AM13L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B6C	CANx_AM13H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B70	CANx_AM14L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B74	CANx_AM14H	"Acceptance Mask Registers (H)" on page 27-53

Table A-9. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2B78	CANx_AM15L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B7C	CANx_AM15H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B80	CANx_AM16L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B84	CANx_AM16H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B88	CANx_AM17L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B8C	CANx_AM17H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B90	CANx_AM18L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B94	CANx_AM18H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2B98	CANx_AM19L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2B9C	CANx_AM19H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2BA0	CANx_AM20L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2BA4	CANx_AM20H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2BA8	CANx_AM21L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0 2BAC	CANx_AM21H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0 2BB0	CANx_AM22L	"Acceptance Mask Registers (L)" on page 27-55

CANx Registers

Table A-9. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0_2BB4	CANx_AM22H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BB8	CANx_AM23L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BBC	CANx_AM23H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BC0	CANx_AM24L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BC4	CANx_AM24H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BC8	CANx_AM25L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BCC	CANx_AM25H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BD0	CANx_AM26L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BD4	CANx_AM26H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BD8	CANx_AM27L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BDC	CANx_AM27H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BE0	CANx_AM28L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BE4	CANx_AM28H	"Acceptance Mask Registers (H)" on page 27-53
0xFFC0_2BE8	CANx_AM29L	"Acceptance Mask Registers (L)" on page 27-55
0xFFC0_2BEC	CANx_AM29H	"Acceptance Mask Registers (H)" on page 27-53

Table A-9. CANx Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2BF0	CANx_AM30L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2BF4	CANx_AM30H	“Acceptance Mask Registers (H)” on page 27-53
0xFFC0 2BF8	CANx_AM31L	“Acceptance Mask Registers (L)” on page 27-55
0xFFC0 2BFC	CANx_AM31H	“Acceptance Mask Registers (H)” on page 27-53

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-10](#) and [Table A-11](#). [Table A-10](#) identifies the base address of each CANx mailbox, as well as the register prefix that identifies mailbox. [Table A-11](#) 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-10. 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_

CANx Registers

Table A-10. CANx Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
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_
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_

Table A-10. CANx Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
29	0xFFC0_2FA0	CANx_MB29_
30	0xFFC0_2FC0	CANx_MB30_
31	0xFFC0_2FE0	CANx_MB31_

Table A-11. CANx Mailbox Register Suffix and Offset

Register Suffix	Offset From Base	See Page
DATA0	0x00	“Mailbox Word 0 Register” on page 27-71
DATA1	0x04	“Mailbox Word 1 Register” on page 27-69
DATA2	0x08	“Mailbox Word 2 Register” on page 27-67
DATA3	0x0C	“Mailbox Word 3 Register” on page 27-65
LENGTH	0x10	“Mailbox Word 4 Register” on page 27-63
TIMESTAMP	0x14	“Mailbox Word 5 Register” on page 27-61
ID0	0x18	“Mailbox Word 6 Register” on page 27-59
ID1	0x1C	“Mailbox Word 7 Register” on page 27-57

SPI0 Controller Registers

SPI0 Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in [Table A-12](#).

Table A-12. SPI0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0500	SPI0_CTL	“SPI Control Register” on page 28-45
0xFFC0 0504	SPI0_FLG	“SPIx Flag Register” on page 28-46
0xFFC0 0508	SPI0_STAT	“SPI Status Register” on page 28-48
0xFFC0 050C	SPI0_TDBR	“SPI Transmit Data Buffer Register” on page 28-48
0xFFC0 0510	SPI0_RDBR	“SPI Receive Data Buffer Register” on page 28-49
0xFFC0 0514	SPI0_BAUD	“SPI Baud Rate Register” on page 28-44
0xFFC0 0518	SPI0_SHADOW	“SPI RDBR Shadow Register” on page 28-49

SPI1 Controller Registers

SPI1 controller registers are listed in [Table A-12](#).

Table A-13. SPI0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2300	SPI1_CTL	“SPI Control Register” on page 28-45
0xFFC0 2304	SPI1_FLG	“SPIx Flag Register” on page 28-46
0xFFC0 2308	SPI1_STAT	“SPI Status Register” on page 28-48

Table A-13. SPI0 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 230C	SPI1_TDBR	"SPI Transmit Data Buffer Register" on page 28-48
0xFFC0 2310	SPI1_RDBR	"SPI Receive Data Buffer Register" on page 28-49
0xFFC0 2314	SPI1_BAUD	"SPI Baud Rate Register" on page 28-44
0xFFC0 2318	SPI1_SHADOW	"SPI RDBR Shadow Register" on page 28-49

TWI Registers

The TWI controller has 16 registers described in the following sections.

[Table A-14](#) lists the TWI registers.

Table A-14. TWIx Registers

TWI0 Memory-mapped Registers	Register Name	Function
0xFFC0 0700	TWIX_CLKDIV	SCL clock divider registers on page 29-37
0xFFC0 0704	TWIX_CONTROL	TWI control registers on page 29-36
0xFFC0 0708	TWIX_SLAVE_CTL	TWI slave mode control registers on page 29-37
0xFFC0 070C	TWIX_SLAVE_ADDR	TWI slave mode address registers on page 29-39
0xFFC0 0710	TWIX_SLAVE_STAT	TWI slave mode status registers on page 29-40
0xFFC0 0714	TWIX_MASTER_CTL	TWI master mode control registers on page 29-41

TWI Registers

Table A-14. TWIx Registers (Cont'd)

TWI0 Memory-mapped Registers	Register Name	Function
0xFFC0 0718	TWIX_MASTER_ADDR	TWI master mode address registers on page 29-44
0xFFC0 071C	TWIX_MASTER_STAT	TWI master mode status registers on page 29-45
0xFFC0 0720	TWIX_INT_STAT	TWI interrupt status registers on page 29-51
0xFFC0 0724	TWIX_INT_MASK	TWI interrupt mask registers on page 29-48
0xFFC0 0728	TWIX_FIFO_CTL	TWI FIFO control registers on page 29-45
0xFFC0 072C	TWIX_FIFO_STAT	TWI FIFO status registers on page 29-47
0xFFC0 0780	TWIX_XMT_DATA8	TWI FIFO transmit data single-byte registers on page 29-52
0xFFC0 0784	TWIX_XMT_DATA16	TWI FIFO transmit data double-byte registers on page 29-53
0xFFC0 0788	TWIX_RCV_DATA8	TWI FIFO receive data single-byte registers on page 29-54
0xFFC0 078C	TWIX_RCV_DATA16	TWI FIFO receive data double-byte registers on page 29-55

SPORT0 Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in [Table A-15](#).

Table A-15. SPORT0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0800	SPORT0_TCR1	“SPORTx Transmit Configuration 1 Register” on page 30-51
0xFFC0 0804	SPORT0_TCR2	“SPORTx Transmit Configuration 2 Register” on page 30-52
0xFFC0 0808	SPORT0_TCLKDIV	“SPORTx Transmit Serial Clock Divider Register” on page 30-68
0xFFC0 080C	SPORT0_TFSDIV	“SPORTx Transmit Frame Sync Divider Register” on page 30-69
0xFFC0 0810	SPORT0_TX	“SPORTx Transmit Data Register” on page 30-63
0xFFC0 0818	SPORT0_RX	“SPORTx Receive Data Register” on page 30-65
0xFFC0 0820	SPORT0_RCR1	“SPORTx Receive Configuration 1 Register” on page 30-57
0xFFC0 0824	SPORT0_RCR2	“SPORTx Receive Configuration 2 Register” on page 30-58
0xFFC0 0828	SPORT0_RCLKDIV	“SPORTx Receive Serial Clock Divider Register” on page 30-68
0xFFC0 082C	SPORT0_RFSDIV	“SPORTx Receive Frame Sync Divider Register” on page 30-69
0xFFC0 0830	SPORT0_STAT	“SPORTx Status Register” on page 30-67
0xFFC0 0834	SPORT0_CHNL	“SPORTx Current Channel Register” on page 30-72

SPORT0 Controller Registers

Table A-15. SPORT0 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0838	SPORT0_MCMC1	“SPORTx Multichannel Configuration Register 1” on page 30-70
0xFFC0 083C	SPORT0_MCMC2	“SPORTx Multichannel Configuration Register 2” on page 30-71
0xFFC0 0840	SPORT0_MTCS0	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0844	SPORT0_MTCS1	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0848	SPORT0_MTCS2	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 084C	SPORT0_MTCS3	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0850	SPORT0_MRCS0	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 0854	SPORT0_MRCS1	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 0858	SPORT0_MRCS2	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 085C	SPORT0_MRCS3	“SPORTx Multichannel Receive Select Registers” on page 30-73

SPORT1 Controller Registers

SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in [Table A-16](#).

Table A-16. SPORT 1 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0900	SPORT1_TCR1	“SPORTx Transmit Configuration 1 Register” on page 30-51
0xFFC0 0904	SPORT1_TCR2	“SPORTx Transmit Configuration 2 Register” on page 30-52
0xFFC0 0908	SPORT1_TCLKDIV	“SPORTx Transmit Serial Clock Divider Register” on page 30-68
0xFFC0 090C	SPORT1_TFSDIV	“SPORTx Transmit Frame Sync Divider Register” on page 30-69
0xFFC0 0910	SPORT1_TX	“SPORTx Transmit Data Register” on page 30-63
0xFFC0 0918	SPORT1_RX	“SPORTx Receive Data Register” on page 30-65
0xFFC0 0920	SPORT1_RCR1	“SPORTx Receive Configuration 1 Register” on page 30-57
0xFFC0 0924	SPORT1_RCR2	“SPORTx Receive Configuration 2 Register” on page 30-58
0xFFC0 0928	SPORT1_RCLKDIV	“SPORTx Receive Serial Clock Divider Register” on page 30-68
0xFFC0 092C	SPORT1_RFSDIV	“SPORTx Receive Frame Sync Divider Register” on page 30-69
0xFFC0 0930	SPORT1_STAT	“SPORTx Status Register” on page 30-67
0xFFC0 0934	SPORT1_CHNL	“SPORTx Current Channel Register” on page 30-72

SPORT1 Controller Registers

Table A-16. SPORT 1 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0938	SPORT1_MCMC1	“SPORTx Multichannel Configuration Register 1” on page 30-70
0xFFC0 093C	SPORT1_MCMC2	“SPORTx Multichannel Configuration Register 2” on page 30-71
0xFFC0 0940	SPORT1_MTCS0	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0944	SPORT1_MTCS1	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0948	SPORT1_MTCS2	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 094C	SPORT1_MTCS3	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 0950	SPORT1_MRCS0	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 0954	SPORT1_MRCS1	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 0958	SPORT1_MRCS2	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 095C	SPORT1_MRCS3	“SPORTx Multichannel Receive Select Registers” on page 30-73

SPORT2 Controller Registers

SPORT2 controller registers are listed in [Table A-17](#).

Table A-17. SPORT2 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2500	SPORT2_TCR1	“SPORTx Transmit Configuration 1 Register” on page 30-51
0xFFC0 2504	SPORT2_TCR2	“SPORTx Transmit Configuration 2 Register” on page 30-52
0xFFC0 2508	SPORT2_TCLKDIV	“SPORTx Transmit Serial Clock Divider Register” on page 30-68
0xFFC0 250C	SPORT2_TFSDIV	“SPORTx Transmit Frame Sync Divider Register” on page 30-69
0xFFC0 2510	SPORT2_TX	“SPORTx Transmit Data Register” on page 30-63
0xFFC0 2518	SPORT2_RX	“SPORTx Receive Data Register” on page 30-65
0xFFC0 2520	SPORT2_RCR1	“SPORTx Receive Configuration 1 Register” on page 30-57
0xFFC0 2524	SPORT2_RCR2	“SPORTx Receive Configuration 2 Register” on page 30-58
0xFFC0 2528	SPORT2_RCLKDIV	“SPORTx Receive Serial Clock Divider Register” on page 30-68
0xFFC0 252C	SPORT2_RFSDIV	“SPORTx Receive Frame Sync Divider Register” on page 30-69
0xFFC0 2530	SPORT2_STAT	“SPORTx Status Register” on page 30-67
0xFFC0 2534	SPORT2_CHNL	“SPORTx Current Channel Register” on page 30-72
0xFFC0 2538	SPORT2_MCMC1	“SPORTx Multichannel Configuration Register 1” on page 30-70

SPORT2 Controller Registers

Table A-17. SPORT2 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 253C	SPORT2_MCMC2	“SPORTx Multichannel Configuration Register 2” on page 30-71
0xFFC0 2540	SPORT2_MTCS0	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2544	SPORT2_MTCS1	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2548	SPORT2_MTCS2	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 254C	SPORT2_MTCS3	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2550	SPORT2_MRCS0	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 2554	SPORT2_MRCS1	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 2558	SPORT2_MRCS2	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 095C	SPORT2_MRCS3	“SPORTx Multichannel Receive Select Registers” on page 30-73

SPORT3 Controller Registers

SPORT3 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in [Table A-18](#).

Table A-18. SPORT3 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2600	SPORT3_TCR1	“SPORTx Transmit Configuration 1 Register” on page 30-51
0xFFC0 2604	SPORT3_TCR2	“SPORTx Transmit Configuration 2 Register” on page 30-52
0xFFC0 2608	SPORT3_TCLKDIV	“SPORTx Transmit Serial Clock Divider Register” on page 30-68
0xFFC0 260C	SPORT3_TFSDIV	“SPORTx Transmit Frame Sync Divider Register” on page 30-69
0xFFC0 2610	SPORT3_TX	“SPORTx Transmit Data Register” on page 30-63
0xFFC0 2618	SPORT3_RX	“SPORTx Receive Data Register” on page 30-65
0xFFC0 2620	SPORT3_RCR1	“SPORTx Receive Configuration 1 Register” on page 30-57
0xFFC0 2624	SPORT3_RCR2	“SPORTx Receive Configuration 2 Register” on page 30-58
0xFFC0 2628	SPORT3_RCLKDIV	“SPORTx Receive Serial Clock Divider Register” on page 30-68
0xFFC0 262C	SPORT3_RFSDIV	“SPORTx Receive Frame Sync Divider Register” on page 30-69
0xFFC0 2630	SPORT3_STAT	“SPORTx Status Register” on page 30-67
0xFFC0 2634	SPORT3_CHNL	“SPORTx Current Channel Register” on page 30-72

SPORT3 Controller Registers

Table A-18. SPORT3 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 2638	SPORT3_MCMC1	“SPORTx Multichannel Configuration Register 1” on page 30-70
0xFFC0 263C	SPORT3_MCMC2	“SPORTx Multichannel Configuration Register 2” on page 30-71
0xFFC0 2640	SPORT3_MTCS0	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2644	SPORT3_MTCS1	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2648	SPORT3_MTCS2	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 264C	SPORT3_MTCS3	“SPORTx Multichannel Transmit Select Registers” on page 30-75
0xFFC0 2650	SPORT3_MRCS0	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 2654	SPORT3_MRCS1	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 2658	SPORT3_MRCS2	“SPORTx Multichannel Receive Select Registers” on page 30-73
0xFFC0 265C	SPORT3_MRCS3	“SPORTx Multichannel Receive Select Registers” on page 30-73

UART0 Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in [Table A-19](#).

Table A-19. UART0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0400	UART0_DLL	“UART Divisor Latch Registers” on page 31-48
0xFFC0 0404	UART0_DLH	“UART Divisor Latch Registers” on page 31-48
0xFFC0 0408	UART0_GCTL	“UART Global Control Registers” on page 31-50
0xFFC0 040C	UART0_LCR	“UART Line Control Registers” on page 31-30
0xFFC0 0410	UART0_MCR	“UART Modem Control Registers” on page 31-33
0xFFC0 0414	UART0_LSR	“UART Line Status Registers” on page 31-36
0xFFC0 0418	UART0_MSR	“UART Modem Control Registers” on page 31-33
0xFFC0 041C	UART0_SCR	“UART Scratch Registers” on page 31-49
0xFFC0 0420	UART0_IER_SET	“UART Interrupt Enable Set Registers” on page 31-44
0xFFC0 0424	UART0_IER_CLEAR	“UART Interrupt Enable Clear Registers” on page 31-45
0xFFC0 0428	UART0_THR	“UART Transmit Holding Registers” on page 31-41
0xFFC0 042C	UART0_RBR	“UART Receive Buffer Registers” on page 31-42

UART1 Controller Registers

UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in [Table A-20](#).

Table A-20. UART1 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0200	UART1_DLL	“UART Divisor Latch Registers” on page 31-48
0xFFC0 0204	UART1_DLH	“UART Divisor Latch Registers” on page 31-48
0xFFC0 0208	UART1_GCTL	“UART Global Control Registers” on page 31-50
0xFFC0 020C	UART1_LCR	“UART Line Control Registers” on page 31-30
0xFFC0 0210	UART1_MCR	“UART Modem Control Registers” on page 31-33
0xFFC0 0214	UART1_LSR	“UART Line Status Registers” on page 31-36
0xFFC0 0218	UART1_MSR	“UART Modem Control Registers” on page 31-33
0xFFC0 021C	UART1_SCR	“UART Scratch Registers” on page 31-49
0xFFC0 0220	UART1_IER_SET	“UART Interrupt Enable Set Registers” on page 31-44
0xFFC0 0224	UART1_IER_CLEAR	“UART Interrupt Enable Clear Registers” on page 31-45
0xFFC0 0228	UART1_THR	“UART Transmit Holding Registers” on page 31-41
0xFFC0 022C	UART1_RBR	“UART Receive Buffer Registers” on page 31-42

UART2 Controller Registers

UART2 controller registers are listed in [Table A-21](#). UART2 is not available on the ADSP-BF542 and ADSP-BF544 processors.

Table A-21. UART2 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2100	UART2_DLL	“UART Divisor Latch Registers” on page 31-48
0xFFC0 2104	UART2_DLH	“UART Divisor Latch Registers” on page 31-48
0xFFC0 2108	UART2_GCTL	“UART Global Control Registers” on page 31-50
0xFFC0 210C	UART2_LCR	“UART Line Control Registers” on page 31-30
0xFFC0 2110	UART2_MCR	“UART Modem Control Registers” on page 31-33
0xFFC0 2114	UART2_LSR	“UART Line Status Registers” on page 31-36
0xFFC0 2118	UART2_MSR	“UART Modem Control Registers” on page 31-33
0xFFC0 211C	UART2_SCR	“UART Scratch Registers” on page 31-49
0xFFC0 2120	UART2_IER_SET	“UART Interrupt Enable Set Registers” on page 31-44
0xFFC0 2124	UART2_IER_CLEAR	“UART Interrupt Enable Clear Registers” on page 31-45
0xFFC0 2128	UART2_THR	“UART Transmit Holding Registers” on page 31-41
0xFFC0 212C	UART2_RBR	“UART Receive Buffer Registers” on page 31-42

UART3 Controller Registers

UART3 controller registers are listed in [Table A-22](#).

Table A-22. UART3 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 3100	UART3_DLL	“UART Divisor Latch Registers” on page 31-48
0xFFC0 3104	UART3_DLH	“UART Divisor Latch Registers” on page 31-48
0xFFC0 3108	UART3_GCTL	“UART Global Control Registers” on page 31-50
0xFFC0 310C	UART3_LCR	“UART Line Control Registers” on page 31-30
0xFFC0 3110	UART3_MCR	“UART Modem Control Registers” on page 31-33
0xFFC0 3114	UART3_LSR	“UART Line Status Registers” on page 31-36
0xFFC0 3118	UART3_MSR	“UART Modem Control Registers” on page 31-33
0xFFC0 311C	UART3_SCR	“UART Scratch Registers” on page 31-49
0xFFC0 3120	UART3_IER_SET	“UART Interrupt Enable Set Registers” on page 31-44
0xFFC0 3124	UART3_IER_CLEAR	“UART Interrupt Enable Clear Registers” on page 31-45
0xFFC0 3128	UART3_THR	“UART Transmit Holding Registers” on page 31-41
0xFFC0 312C	UART3_RBR	“UART Receive Buffer Registers” on page 31-42

USB OTG Registers

Descriptions and bit diagrams for each of these MMRs are provided in the following sections. See [Table A-23](#).

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers

Memory-mapped Address	Name	Function
0xFFC03C00	USB_FADDR	USB function address register on page 32-81
0xFFC03C04	USB_POWER	USB power management register on page 32-82
0xFFC03C08	USB_INTRTX	USB transmit interrupt register on page 32-85 for endpoint 0 and Tx endpoint 1 to 7
0xFFC03C0C	USB_INTRRX	USB receive interrupt register on page 32-85 for Rx endpoints 1 to 7
0xFFC03C10	USB_INTRTXE	USB transmit interrupt enable register on page 32-86 for IntrTx
0xFFC03C14	USB_INTRRXE	USB receive interrupt enable register on page 32-88 for IntrRx
0xFFC03C18	USB_INTRUSB	USB common interrupt register on page 32-89
0xFFC03C1C	USB_INTRUSBE	USB common interrupt enable register on page 32-90
0xFFC03C20	USB_FRAME	USB frame number register on page 32-91
0xFFC03C24	USB_INDEX	USB index register on page 32-91
0xFFC03C28	USB_TESTMODE	USB test mode register on page 32-93 <i>(for Analog Devices internal use only)</i>
0xFFC03C2C	USB_GLOBINTR	USB global interrupt register on page 32-94
0xFFC03C30	USB_GLOBAL_CTL	USB global control register on page 32-95

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Packet Control – Indexed Registers		
0xFFC03C40	USB_TX_MAX_PACKET	USB Tx maximum packet register on page 32-97
0xFFC03C44	USB_CSR0	USB control/status register on page 32-98
0xFFC03C44	USB_TXCSR	USB Tx control/status EPx register on page 32-102
0xFFC03C48	USB_RX_MAX_PACKET	USB Rxx maximum packet register on page 32-107
0xFFC03C4C	USB_RXCSR	USB Rx control/status EPx register on page 32-109t
0xFFC03C50	USB_COUNT0	USB count 0 register on page 32-115
0xFFC03C50	USB_RXCOUNT	USB Rx byte count EPx register on page 32-116
0xFFC03C54	USB_TXTYPE	USB Tx type register on page 32-117
0xFFC03C58	USB_NAKLIMIT0	USB NAK 0 limit register on page 32-117
0xFFC03C58	USB_TXINTERVAL	USB Tx interval register on page 32-118
0xFFC03C5C	USB_RXTYPE	USB Rx type register on page 32-119
0xFFC03C60	USB_RXINTERVAL	USB Rx interval register on page 32-120
0xFFC03C68	USB_TXCOUNT	USB Tx byte count EPx register on page 32-121

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint FIFO Registers		
0xFFC03C80	USB_EP0_FIFO	USB endpoint FIFO 0 register
0xFFC03C88	USB_EP1_FIFO	USB endpoint FIFO 1 register
0xFFC03C90	USB_EP2_FIFO	USB endpoint FIFO 2 register
0xFFC03C98	USB_EP3_FIFO	USB endpoint FIFO 30 register
0xFFC03CA0	USB_EP4_FIFO	USB endpoint FIFO 4 register
0xFFC03CA8	USB_EP5_FIFO	USB endpoint FIFO 5 register
0xFFC03CB0	USB_EP6_FIFO	USB endpoint FIFO 6 register
0xFFC03CB8	USB_EP7_FIFO	USB endpoint FIFO 7 register
USB OTG Control Registers		
0xFFC03D00	USB_OTG_DEV_CTL	USB OTG device control register on page 32-122
0xFFC03D04	USB_OTG_VBUS_IRQ	USB OTG VBUS interrupt register on page 32-124
0xFFC03D08	USB_OTG_VBUS_MASK	USB VBUS mask register on page 32-126

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB PHY Control Registers		
0xFFC03D48	USB_LINKINFO	USB link info register on page 32-127
0xFFC03D4C	USB_VPLEN	USB VBUS pulse length register on page 32-127
0xFFC03D50	USB_HS_EOF1	USB high-speed EOF 1 register on page 32-128
0xFFC03D54	USB_FS_EOF1	USB full-speed EOF 1 register on page 32-128
0xFFC03D58	USB_LS_EOF1	USB low-speed EOF 1 register on page 32-129
0xFFC03DE0	USB_APHY_CNTRL	USB APHY control 2 register on page 32-130 <i>(for Analog Devices internal use only)</i>
0xFFC03DE4	USB_APHY_CALIB	USB APHY calibration register <i>(for Analog Devices internal use only)</i>
0xFFC03DE8	USB_APHY_CNTRL2	Used to prevent re-enumeration after the processor goes into hibernate mode
0xFFC03DEC	USB_PHY_TEST	Register used for PHY and FIFO test features <i>(for Analog Devices internal use only)</i>
0xFFC03DF0	USB_PLLOSC_CTRL	USB PLL OSC control register on page 32-132
0xFFC03DF4	USB_SRP_CLKDIV	USB SRP clock divider register on page 32-133

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 0 Control Registers		
0xFFC03E00	USB_EP_NI0_TXMAXP	Maximum packet size for host Tx endpoint0
0xFFC03E04	USB_EP_NI0_TXCSR	Control Status register for endpoint 0
0xFFC03E08	USB_EP_NI0_RXMAXP	Maximum packet size for host Rx endpoint0
0xFFC03E0C	USB_EP_NI0_RXCSR	Control Status register for host Rx endpoint0
0xFFC03E10	USB_EP_NI0_RXCOUNT	Number of bytes received in endpoint 0 FIFO
0xFFC03E14	USB_EP_NI0_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint0
0xFFC03E18	USB_EP_NI0_TXINTERVAL	Sets the NAK response timeout on endpoint 0
0xFFC03E1C	USB_EP_NI0_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint0
0xFFC03E20	USB_EP_NI0_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint0

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 1 Control Registers		
0xFFC03E28	USB_EP_NI0_TXCOUNT	Number of bytes to be written to the endpoint0 Tx FIFO
0xFFC03E40	USB_EP_NI1_TXMAXP	Maximum packet size for host Tx endpoint1
0xFFC03E44	USB_EP_NI1_TXCSR	Control Status register for endpoint1
0xFFC03E48	USB_EP_NI1_RXMAXP	Maximum packet size for host Rx endpoint1
0xFFC03E4C	USB_EP_NI1_RXCSR	Control Status register for host Rx endpoint1
0xFFC03E50	USB_EP_NI1_RXCOUNT	Number of bytes received in endpoint1 FIFO
0xFFC03E54	USB_EP_NI1_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint1
0xFFC03E58	USB_EP_NI1_TXINTERVAL	Sets the NAK response timeout on endpoint1
0xFFC03E5C	USB_EP_NI1_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint1
0xFFC03E60	USB_EP_NI1_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint1

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 2 Control Registers		
0xFFC03E68	USB_EP_NI1_TXCOUNT	Number of bytes to be written to the+H102 endpoint1 Tx FIFO
0xFFC03E80	USB_EP_NI2_TXMAXP	Maximum packet size for host Tx endpoint2
0xFFC03E84	USB_EP_NI2_TXCSR	Control Status register for endpoint2
0xFFC03E88	USB_EP_NI2_RXMAXP	Maximum packet size for host Rx endpoint2
0xFFC03E8C	USB_EP_NI2_RXCSR	Control Status register for host Rx endpoint2
0xFFC03E90	USB_EP_NI2_RXCOUNT	Number of bytes received in endpoint2 FIFO
0xFFC03E94	USB_EP_NI2_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint2
0xFFC03E98	USB_EP_NI2_TXINTERVAL	Sets the NAK response timeout on endpoint2
0xFFC03E9C	USB_EP_NI2_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint2
0xFFC03EA0	USB_EP_NI2_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint2

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 3 Control Registers		
0xFFC03EA8	USB_EP_NI2_TXCOUNT	Number of bytes to be written to the endpoint2 Tx FIFO
0xFFC03EC0	USB_EP_NI3_TXMAXP	Maximum packet size for host Tx endpoint3
0xFFC03EC4	USB_EP_NI3_TXCSR	Control Status register for endpoint3
0xFFC03EC8	USB_EP_NI3_RXMAXP	Maximum packet size for host Rx endpoint3
0xFFC03ECC	USB_EP_NI3_RXCSR	Control Status register for host Rx endpoint3
0xFFC03ED0	USB_EP_NI3_RXCOUNT	Number of bytes received in endpoint3 FIFO
0xFFC03ED4	USB_EP_NI3_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint3
0xFFC03ED8	USB_EP_NI3_TXINTERVAL	Sets the NAK response timeout on endpoint3
0xFFC03EDC	USB_EP_NI3_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint3
0xFFC03EE0	USB_EP_NI3_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint3

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 4 Control Registers		
0xFFC03EE8	USB_EP_NI3_TXCOUNT	Number of bytes to be written to the H124endpoint3 Tx FIFO
0xFFC03F00	USB_EP_NI4_TXMAXP	Maximum packet size for host Tx endpoint4
0xFFC03F04	USB_EP_NI4_TXCSR	Control Status register for endpoint4
0xFFC03F08	USB_EP_NI4_RXMAXP	Maximum packet size for host Rx endpoint4
0xFFC03F0C	USB_EP_NI4_RXCSR	Control Status register for host Rx endpoint4
0xFFC03F10	USB_EP_NI4_RXCOUNT	Number of bytes received in endpoint4 FIFO
0xFFC03F14	USB_EP_NI4_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint4
0xFFC03F18	USB_EP_NI4_TXINTERVAL	Sets the NAK response timeout on endpoint4
0xFFC03F1C	USB_EP_NI4_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint4
0xFFC03F20	USB_EP_NI4_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint4

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 5 Control Registers		
0xFFC03F28	USB_EP_NI4_TXCOUNT	Number of bytes to be written to the endpoint4 Tx FIFO
0xFFC03F40	USB_EP_NI5_TXMAXP	Maximum packet size for host Tx endpoint5
0xFFC03F44	USB_EP_NI5_TXCSR	Control Status register for endpoint5
0xFFC03F48	USB_EP_NI5_RXMAXP	Maximum packet size for host Rx endpoint5
0xFFC03F4C	USB_EP_NI5_RXCSR	Control Status register for host Rx endpoint5
0xFFC03F50	USB_EP_NI5_RXCOUNT	Number of bytes received in endpoint5 FIFO
0xFFC03F54	USB_EP_NI5_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint5
0xFFC03F58	USB_EP_NI5_TXINTERVAL	Sets the NAK response timeout on endpoint5
0xFFC03F5C	USB_EP_NI5_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint5
0xFFC03F60	USB_EP_NI5_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint5

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 6 Control Registers		
0xFFC03F68	USB_EP_NI5_TXCOUNT	Number of bytes to be written to the H145endpoint5 Tx FIFO
0xFFC03F80	USB_EP_NI6_TXMAXP	Maximum packet size for host Tx endpoint6
0xFFC03F84	USB_EP_NI6_TXCSR	Control Status register for endpoint6
0xFFC03F88	USB_EP_NI6_RXMAXP	Maximum packet size for host Rx endpoint6
0xFFC03F8C	USB_EP_NI6_RXCSR	Control Status register for host Rx endpoint6
0xFFC03F90	USB_EP_NI6_RXCOUNT	Number of bytes received in endpoint6 FIFO
0xFFC03F94	USB_EP_NI6_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint6
0xFFC03F98	USB_EP_NI6_TXINTERVAL	Sets the NAK response timeout on endpoint6
0xFFC03F9C	USB_EP_NI6_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint6
0xFFC03FA0	USB_EP_NI6_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint6

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Endpoint 7 Control Registers		
0xFFC03FA8	USB_EP_NI6_TXCOUNT	Number of bytes to be written to the endpoint6 Tx FIFO
0xFFC03FC0	USB_EP_NI7_TXMAXP	Maximum packet size for host Tx endpoint7
0xFFC03FC4	USB_EP_NI7_TXCSR	Control Status register for endpoint7
0xFFC03FC8	USB_EP_NI7_RXMAXP	Maximum packet size for host Rx endpoint7
0xFFC03FCC	USB_EP_NI7_RXCSR	Control Status register for host Rx endpoint7
0xFFC03FD0	USB_EP_NI7_RXCOUNT	Number of bytes received in endpoint7 FIFO
0xFFC03FD4	USB_EP_NI7_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Tx endpoint7
0xFFC03FD8	USB_EP_NI7_TXINTERVAL	Sets the NAK response timeout on endpoint7
0xFFC03FDC	USB_EP_NI7_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host Rx endpoint7
0xFFC03FF0	USB_EP_NI7_RXINTERVAL	Sets the polling interval for interrupt and isochronous transfers or the NAK response timeout on bulk transfers for host Rx endpoint7
0xFFC03FF8	USB_EP_NI7_TXCOUNT	Number of bytes to be written to the endpoint7 Tx FIFO
USB DMA Registers		
0xFFC04000	USB_DMA_INTERRUPT	USB DMA interrupt register on page 32-134

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Channel 0 Config Registers		
0xFFC04004	USB_DMA0CONTROL	USB DMA control register 0 on page 32-135
0xFFC04008	USB_DMA0ADDR LOW	USB DMA address low register 0 on page 32-137
0xFFC0400C	USB_DMA0ADDR HIGH	USB DMA address high register 0 on page 32-138
0xFFC04010	USB_DMA0COUNT LOW	USB DMA count low register 0 on page 32-139
0xFFC04014	USB_DMA0COUNT HIGH	USB DMA count high register 0 on page 32-140
USB Channel 1 Config Registers		
0xFFC04024	USB_DMA1CONTROL	USB DMA control register 1 on page 32-135
0xFFC04028	USB_DMA1ADDR LOW	USB DMA address low register 1 on page 32-137
0xFFC0402C	USB_DMA1ADDR HIGH	USB DMA address high register 1 on page 32-138
0xFFC04030	USB_DMA1COUNT LOW	USB DMA count low register 1 on page 32-139
0xFFC04034	USB_DMA1COUNT HIGH	USB DMA count high register 1 on page 32-140

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Channel 2 Config Registers		
0xFFC04044	USB_DMA2CONTROL	USB DMA control register 2 on page 32-135
0xFFC04048	USB_DMA2ADDR LOW	USB DMA address low register 2 on page 32-137
0xFFC0404C	USB_DMA2ADDR HIGH	USB DMA address high register 2 on page 32-138
0xFFC04050	USB_DMA2COUNT LOW	USB DMA count low register 2 on page 32-139
0xFFC04054	USB_DMA2COUNT HIGH	USB DMA count high register 2 on page 32-140
USB Channel 3 Config Registers		
0xFFC04064	USB_DMA3CONTROL	USB DMA control register 3 on page 32-135
0xFFC04068	USB_DMA3ADDR LOW	USB DMA address low register 3 on page 32-137
0xFFC0406C	USB_DMA3ADDR HIGH	USB DMA address high register 3 on page 32-138
0xFFC04070	USB_DMA3COUNT LOW	USB DMA count low register 3 on page 32-139
0xFFC04074	USB_DMA3COUNT HIGH	USB DMA count high register 3 on page 32-140

USB OTG Registers

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Channel 4 Config Registers		
0xFFC04084	USB_DMA4CONTROL	USB DMA control register 4 on page 32-135
0xFFC04088	USB_DMA4ADDR LOW	USB DMA address low register 4 on page 32-137
0xFFC0408C	USB_DMA4ADDR HIGH	USB DMA address high register 4 on page 32-138
0xFFC04090	USB_DMA4COUNT LOW	USB DMA count low register 4 on page 32-139
0xFFC04094	USB_DMA4COUNT HIGH	USB DMA count high register 4 on page 32-140
USB Channel 5 Config Registers		
0xFFC040A4	USB_DMA5CONTROL	USB DMA control register 5 on page 32-135
0xFFC040A8	USB_DMA5ADDR LOW	USB DMA address low register 5 on page 32-137
0xFFC040AC	USB_DMA5ADDR HIGH	USB DMA address high register 5 on page 32-138
0xFFC040B0	USB_DMA5COUNT LOW	USB DMA count low register 5 on page 32-139
0xFFC040B4	USB_DMA5COUNT HIGH	USB DMA count high register 5 on page 32-140

Table A-23. USB OTG Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Name	Function
USB Channel 6 Config Registers		
0xFFC040C4	USB_DMA6CONTROL	USB DMA control register 6 on page 32-135
0xFFC040C8	USB_DMA6ADDR LOW	USB DMA address low register 6 on page 32-137
0xFFC040CC	USB_DMA6ADDR HIGH	USB DMA address high register 6 on page 32-138
0xFFC040D0	USB_DMA6COUNT LOW	USB DMA count low register 6 on page 32-139
0xFFC040D4	USB_DMA6COUNT HIGH	USB DMA count high register 6 on page 32-140
USB Channel 7 Config Registers		
0xFFC040E4	USB_DMA7CONTROL	USB DMA control register 7 on page 32-135
0xFFC040E8	USB_DMA7ADDR LOW	USB DMA address low register 7 on page 32-137
0xFFC040EC	USB_DMA7ADDR HIGH	USB DMA address high register 7 on page 32-138
0xFFC040F0	USB_DMA7COUNT LOW	USB DMA count low register 7 on page 32-139
0xFFC040F4	USB_DMA7COUNT HIGH	USB DMA count high register 7 on page 32-140

USB OTG Registers

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-3](#)

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

JTAG Standard

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 the “Security” chapter in the ADSP-BF54x *Blackfin Processor Hardware Reference*.

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{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.

Boundary-Scan Architecture

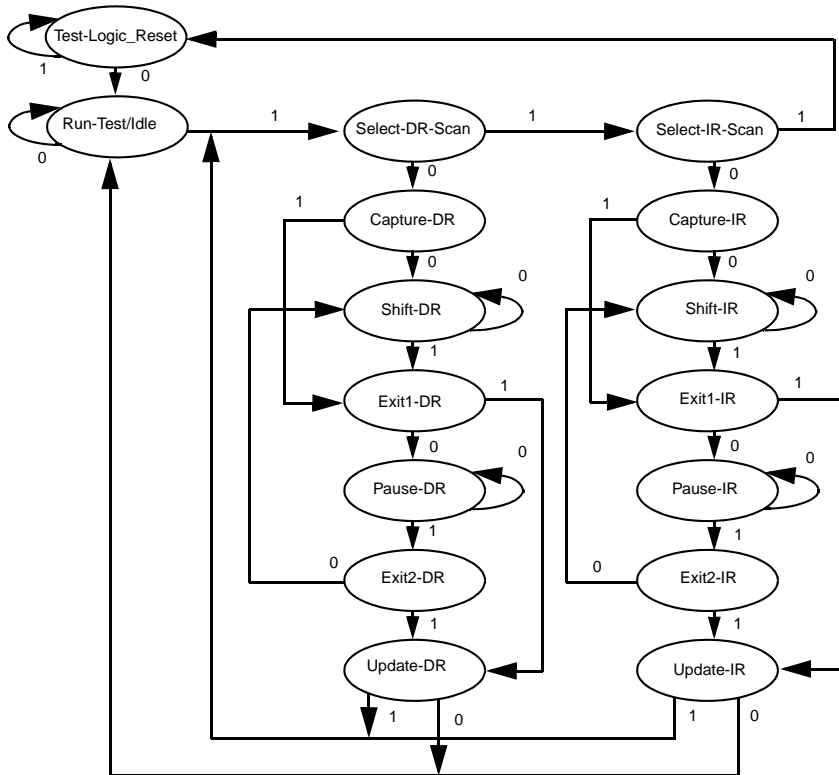


Figure B-1. TAP Controller State Diagram

Note:

- The TAP controller enters the test-logic-reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the test-logic-reset state when $\overline{\text{TRST}}$ 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.

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](#).

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.

Boundary-Scan Architecture

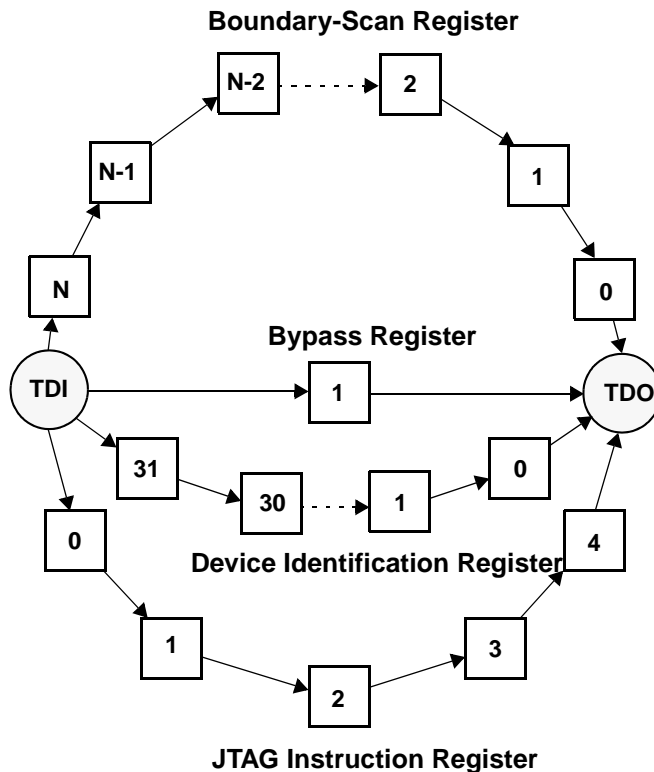


Figure B-2. Serial Scan Paths

The `EXTTEST` instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.



To protect the internal logic when the boundary outputs are over-driven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The `SAMPLE/PRELOAD` instruction performs two functions and selects the Boundary-Scan register to be connected between `TDI` and `TDO`. The instruction has no effect on internal logic.

The `SAMPLE` part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of `TCK`.

The `PRELOAD` part of the instruction allows data to be loaded on the device pins and driven out on the board with the `EXTEST` instruction. Data is preloaded on the pins on the falling edge of `TCK`.

BYPASS – Binary Code 11111

The `BYPASS` instruction selects the `BYPASS` register to be connected to `TDI` and `TDO`. The instruction has no effect on the internal logic. No data inversion should occur between `TDI` and `TDO`.

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.

Boundary-Scan Architecture

I INDEX

Symbols

, 21-69

'A' or 'B' device indicator (B_DEVICE)
bit, 32-122, 32-124

'B' or 'A' device indicator (B_DEVICE)
bit, 32-122, 32-124

μ-law companding, 30-26, 30-31

A

AAIF bit, 27-27, 27-52

AAIM bit, 27-27, 27-50

AAIS bit, 27-27, 27-51

AA_n bit, 27-80

ABO bit, 27-46

abort acknowledge interrupt, CAN, 27-27

acceptance mask filtering, CAN, 27-18

acceptance mask register (CAN_AM_{xx}H),
27-53

acceptance mask register (CAN_AM_{xx}L),
27-55

access denied interrupt, CAN, 27-27

access to unimplemented address interrupt,
CAN, 27-27

ACKE bit, 27-89

active low/high frame syncs, serial port,
30-35

active mode, 20-33

Active Mode (ACTIVE) bit, 21-13

ACTS bit, 31-33

ADCs, connecting to, 30-2

ADIF bit, 27-27, 27-52

ADIM bit, 27-27, 27-50

ADIS bit, 27-27, 27-51

advanced technology attachment packet
interface, 24-1

A-law companding, 30-26, 30-31

All Bypass-MXVR Disabled Mode, 21-12

Allocation Table, 21-57

Allocation Table Updated (ATU) interrupt
event, 21-33

Allocation Table Updated interrupt enable,
21-45

alternate frame sync mode, 30-38

alternate timing, serial port, 30-37

AMC, 20-15

AME bit, 27-57

AMIDE bit, 27-53

ANAK bit, 29-15, 29-45

AP Data field, 21-129

AP Destination Address, 21-128

application data, loading, 20-35

AP Priority, 21-128

APRCEEN, 21-48

APREN, 21-48

APROFEN, 21-48

APRPEEN, 21-48

AP Source Address, 21-129

APTCEN, 21-48

APTSEN, 21-48

arbitration

 TWI, 29-8

ARTS bit, 31-33

asynchronous controller, 20-15

Index

- Asynchronous Packet Arbitrating (APARB) bit, [21-22](#)
- Asynchronous Packet Continuation (APCONT) bit, [21-27](#)
- Asynchronous Packet Receive Buffer (APRB), [21-77](#)
- Asynchronous Packet Receive Buffer Entry Field Offsets, [21-130](#)
- Asynchronous Packet Receive Buffer Entry x (APRBEx) bits, [21-79](#)
- Asynchronous Packet Receive Buffer Overflow (APROF) interrupt event, [21-41](#)
- Asynchronous Packet Receive Buffer Overflow interrupt enable, [21-45](#), [21-48](#)
- Asynchronous Packet Receive CRC Error (APRCE) interrupt event, [21-42](#)
- Asynchronous Packet Receive CRC Error interrupt enable, [21-45](#), [21-48](#)
- Asynchronous Packet Received (APR) bit, [21-23](#)
- Asynchronous Packet Received (APR) interrupt event, [21-41](#)
- Asynchronous Packet Received interrupt enable, [21-48](#)
- Asynchronous Packet Receive Enable (APRXEN) bit, [21-17](#)
- Asynchronous Packet Receive Packet Error (APRPE) interrupt event, [21-42](#)
- Asynchronous Packet Receive Packet Error interrupt enable, [21-46](#), [21-48](#)
- Asynchronous Packet Receiving (APRX) bit, [21-23](#)
- Asynchronous Packet Reception, [21-129](#)
- Asynchronous Packet Transmission, [21-126](#)
- Asynchronous Packet Transmit Buffer (APTb), [21-77](#)
- Asynchronous Packet Transmit Buffer Busy (APBSY) bit, [21-22](#)
- Asynchronous Packet Transmit Buffer Field Offsets, [21-127](#)
- Asynchronous Packet Transmit Buffer Successfully Cancelled (APTC) interrupt event, [21-42](#)
- Asynchronous Packet Transmit Buffer Successfully Cancelled interrupt enable, [21-48](#)
- Asynchronous Packet Transmit Buffer Successfully Sent (APTS) interrupt event, [21-42](#)
- Asynchronous Packet Transmit Buffer Successfully Sent interrupt enable, [21-48](#)
- Asynchronous Packet Transmitting (APTx) bit, [21-22](#)
- asynchronous serial communications, [31-6](#)
- ATA interface, [24-1](#)
- ATAPI
 - ATAPI Signals Summary, [24-3](#)
 - host DMA state M=machine, [24-13](#)
 - host ultra DMA command protocol transfers, [24-16](#)
 - PIO data-In state machine, [24-10](#)
 - PIO data-put protocol state machine, [24-7](#)
 - power-on and hardware reset protocol, [24-20](#)
 - summary of IDE/ATA standards, [24-78](#)
- ATAPI_ADDR (ATAPI address line status) bits, [24-60](#)
- ATAPI address line status (ATAPI_ADDR) bits, [24-60](#)
- ATAPI chip select 0 line status (ATAPI_CS0N) bit, [24-60](#)
- ATAPI chip select 1 line status (ATAPI_CS1N) bit, [24-60](#)

- ATAPI_CONTROL (ATAPI control) register, [24-48](#), [24-50](#), [A-16](#)
- ATAPI_CS0N (ATAPI chip select 0 line status) bit, [24-60](#)
- ATAPI_CS1N (ATAPI chip select 1 line status) bit, [24-60](#)
- ATAPI_DASP (device DASP to host line status) bit, [24-60](#)
- ATAPI_DEV_ADDR (ATAPI device register address) register, [24-48](#), [24-53](#), [A-16](#)
- ATAPI device I/O registers, [24-69](#)
- ATAPI device register address (ATAPI_DEV_ADDR) register, [24-48](#), [24-53](#), [A-16](#)
- ATAPI device register receive data (ATAPI_DEV_RXBUF) register, [24-48](#), [24-55](#), [A-16](#)
- ATAPI device register write data (ATAPI_DEV_TXBUF) register, [24-48](#), [24-54](#), [A-16](#)
- ATAPI_DEV_INT (device interrupt status) bit, [24-58](#)
- ATAPI_DEV_INT_MASK (device interrupt mask) bit, [24-56](#)
- ATAPI_DEV_RXBUF (ATAPI device register receive data) register, [24-48](#), [24-55](#), [A-16](#)
- ATAPI_DEV_TXBUF (ATAPI device register write data) register, [24-48](#), [24-54](#), [A-16](#)
- ATAPI_DIORN (ATAPI read line status) bit, [24-60](#)
- ATAPI_DIOWN (ATAPI write line status) bit, [24-60](#)
- ATAPI_DMAACKN (ATAPI DMA acknowledge line status) bit, [24-60](#)
- ATAPI DMA acknowledge line status (ATAPI_DMAACKN) bit, [24-60](#)
- ATAPI_DMAREQ (ATAPI DMA request line status) bit, [24-60](#)
- ATAPI DMA request line status (ATAPI_DMAREQ) bit, [24-60](#)
- ATAPI_DMA_TFRCNT (ATAPI DMA transfer count) register, [24-49](#), [24-63](#), [A-16](#)
- ATAPI DMA transfer count (ATAPI_DMA_TFRCNT) register, [24-49](#), [24-63](#), [A-16](#)
- ATAPI_HOST_TERM (host termination) bit, [24-61](#)
- ATAPI host terminate (ATAPI_TERMINATE) register, [24-48](#), [24-61](#), [A-16](#)
- ATAPI interface, [24-1](#)
- ATAPI interrupt mask (ATAPI_INT_MASK) register, [24-48](#), [24-56](#), [A-16](#)
- ATAPI interrupt status (ATAPI_INT_STATUS) register, [24-48](#), [24-58](#), [A-16](#)
- ATAPI_INT_MASK (ATAPI interrupt mask) register, [24-48](#), [24-56](#), [A-16](#)
- ATAPI_INTR (device interrupt to host line status) bit, [24-60](#)
- ATAPI_INT_STATUS (ATAPI interrupt status) register, [24-48](#), [24-58](#), [A-16](#)
- ATAPI_IORDY (ATAPI IORDY line status) bit, [24-60](#)
- ATAPI IORDY line status (ATAPI_IORDY) bit, [24-60](#)
- ATAPI IORDY line status (UDMAOUT_CSTATE) bits, [24-61](#)
- ATAPI_LINE_STATUS (ATAPI line status) register, [24-48](#), [24-60](#), [A-16](#)
- ATAPI line status (ATAPI_LINE_STATUS) register, [24-48](#), [24-60](#), [A-16](#)

Index

- ATAPI MDMA timing 0
 - (ATAPI_MULTI_TIM_0) register, [24-49](#), [24-66](#), [A-17](#)
- ATAPI MDMA timing 1
 - (ATAPI_MULTI_TIM_1) register, [24-49](#), [24-66](#), [A-17](#)
- ATAPI MDMA timing 2
 - (ATAPI_MULTI_TIM_2) register, [24-49](#), [24-67](#), [A-17](#)
- ATAPI_MULTI_TIM_0 (ATAPI MDMA timing 0) register, [24-49](#), [24-66](#), [A-17](#)
- ATAPI_MULTI_TIM_1 (ATAPI MDMA timing 1) register, [24-49](#), [24-66](#), [A-17](#)
- ATAPI_MULTI_TIM_2 (ATAPI MDMA timing 2) register, [24-49](#), [24-67](#), [A-17](#)
- ATAPI_PIO_TFRCNT (ATAPI PIO transfer count) register, [24-48](#), [24-62](#), [A-16](#)
- ATAPI_PIO_TIM_0 (ATAPI PIO timing 0) register, [24-49](#), [24-65](#), [A-17](#)
- ATAPI_PIO_TIM_1 (ATAPI PIO timing 1) register, [24-49](#), [24-65](#), [A-17](#)
- ATAPI PIO timing 0
 - (ATAPI_PIO_TIM_0) register, [24-49](#), [24-65](#), [A-17](#)
- ATAPI PIO timing 1
 - (ATAPI_PIO_TIM_1) register, [24-49](#), [24-65](#), [A-17](#)
- ATAPI PIO transfer count
 - (ATAPI_PIO_TFRCNT) register, [24-48](#), [24-62](#), [A-16](#)
- ATAPIPI status (ATAPIPI_STATUS) register, [24-63](#)
- ATAPI read line status (ATAPI_DIORN) bit, [24-60](#)
- ATAPI registers, [24-48](#), [A-16](#)
- ATAPI register transfer timing 0
 - (ATAPI_REG_TIM_0) register, [24-49](#), [24-64](#), [A-17](#)
- ATAPI_SM_STATE (ATAPI state machine status) register, [24-48](#), [24-61](#), [A-16](#)
- ATAPI standards reference, [24-74](#)
- ATAPI state machine status
 - (ATAPI_SM_STATE) register, [24-48](#), [24-61](#), [A-16](#)
- ATAPI_STATUS (ATAPI status) register, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- ATAPI status (ATAPI_STATUS) register, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- ATAPI_STATUS register, [24-48](#), [24-52](#), [A-16](#)
- ATAPI_TERMINATE (ATAPI host terminate) register, [24-48](#), [24-61](#), [A-16](#)
- ATAPI transfer length
 - (ATAPI_XFER_LEN) register, [24-48](#), [24-59](#), [A-16](#)
- ATAPI_UDMAIN_TFRCNT (ATAPI UDMA transfer count) register, [24-49](#), [24-63](#), [A-16](#)
- ATAPI_UDMAOUT_TFRCNT (ATAPI UDMAOUT transfer count) register, [24-49](#), [24-64](#), [A-17](#)
- ATAPI UDMAOUT transfer count
 - (ATAPI_UDMAOUT_TFRCNT) register, [24-49](#), [24-64](#), [A-17](#)
- ATAPI UDMA timing 0
 - (ATAPI_ULTRA_TIM_0) register, [24-49](#), [24-67](#), [A-17](#)

ATAPI UDMA timing 1
 (ATAPI_ULTRA_TIM_1) register,
 24-49, 24-68, A-17
 ATAPI UDMA timing 2
 (ATAPI_ULTRA_TIM_2) register,
 24-49, 24-68, A-17
 ATAPI UDMA timing 3
 (ATAPI_ULTRA_TIM_3) register,
 24-49, 24-69, A-17
 ATAPI UDMA transfer count
 (ATAPI_UDMAIN_TFRCNT)
 register, 24-49, 24-63, A-16
 ATAPI_ULTRA_TIM_0 (ATAPI UDMA
 timing 0) register, 24-49, 24-67, A-17
 ATAPI_ULTRA_TIM_1 (ATAPI UDMA
 timing 1) register, 24-49, 24-68, A-17
 ATAPI_ULTRA_TIM_2 (ATAPI UDMA
 timing 2) register, 24-49, 24-68, A-17
 ATAPI_ULTRA_TIM_3 (ATAPI UDMA
 timing 3) register, 24-49, 24-69, A-17
 ATAPI write line status
 (ATAPI_DIOWN) bit, 24-60
 ATAPI_XFER_LEN (ATAPI transfer
 length) register, 24-48, 24-59, A-16
 ATUEN, 21-45
 autobaud detection, 31-21
 Autobuffer Mode, 21-63, 21-71, 21-74,
 21-76
 AUTOCLEAR_R (RxPktRdy autoclear
 enable) bit, 32-109
 AUTOREQ_RH (autoset ReqPkt) bit,
 32-109
 autoset ReqPkt (AUTOREQ_R) bit,
 32-109
 AUTOSSET_T (TxPktRdy autoset enable)
 bit, 32-102
 auto-transmit mode, CAN, 27-16

B

bable or reset indicator
 (RESET_OR_BABLE_B) bit, 32-89
 bable or reset IRQ enable
 (RESET_OR_BABLE_BE) bit,
 32-90
 BASEID[10:0] field, 27-53, 27-57
 baud rate
 SPI, 28-23
 UART, 31-9, 31-20
 baud rate[15:0] field, 28-44
 BAUD_RATE (baud rate) bits, 28-44
 baud rate (BAUD_RATE) bits, 28-44
 BCZEN, 21-46
 B_DEVICE ('A' or 'B' device indicator)
 bit, 32-122, 32-124
 BEF bit, 27-89
 BI (break indicator) bit, 31-37
 BI (break interrupt) bit, 31-36
 binary decode, B-5
 Biphase Mark Coding Error (BMERR)
 interrupt event, 21-39
 Biphase Mark Coding Error interrupt
 enable, 21-46
 bit order, selecting, 30-30
 BL2UEN, 21-45
 Blackfin processor family
 I/O memory space, 20-10
 memory architecture, 20-6
 BLANKGEN (ITU output with internal
 blanking) bit, 26-82
 Block Counter Zero (BCZ) interrupt event,
 21-39
 Block Counter Zero interrupt enable,
 21-46

Index

block diagrams

- CAN, [27-4](#)
 - EPPI, [26-5](#)
 - processor, [20-5](#)
 - SPI, [28-3](#)
 - SPORT, [30-7](#)
 - TWI, [29-3](#)
 - UART, [31-3](#), [31-12](#)
- Block Locked (BLOCK) bit, [21-25](#)
- Block Locked to Unlocked (BL2U)
interrupt event, [21-35](#)
- Block Unlocked to Locked (BU2L)
interrupt event, [21-35](#)
- BMERREN, [21-46](#)
- BOIF bit, [27-28](#), [27-52](#)
- BOIM bit, [27-28](#), [27-50](#)
- BOIS bit, [27-28](#), [27-51](#)
- boost PLL amplitude (TM_PLL_VCO)
bit, [32-132](#)
- booting, [20-35](#) to ??
- boot kernel, [20-35](#)
- boot ROM
internal, [20-35](#)
- boot stream, [20-35](#)
- boundary-scan architecture, [B-3](#)
- boundary-scan register, [B-7](#)
- broadcast mode, [28-12](#), [28-19](#), [28-20](#)
- BRP[9:0] field, [27-11](#), [27-48](#)
- BU2LEN, [21-45](#)
- Buffer start address register, Initialization,
[21-120](#)
- BUFRDERR bit, [29-14](#), [29-45](#)
- BUFWRERR bit, [29-14](#), [29-45](#)
- BURST_MODE (DMA burst mode
selection) bits, [32-135](#)
- BUSBUSY bit, [29-13](#), [29-45](#)
- BUSERROR (DMA bus error) bit, [32-135](#)
- buses
bandwidth, [20-4](#)
and peripherals, [20-4](#)

- bus-off interrupt, CAN, [27-28](#)
- bus standard, I²C, [20-16](#)
- bypass divisor (CLKDIV_BYPASS) bit,
[23-23](#)
- BYPASS instruction, [B-7](#)
- bypass register, [B-7](#)

C

- CAN, [27-1](#) to [27-97](#)
- acceptance mask filtering, [27-18](#)
 - acceptance mask registers, [27-6](#)
 - acknowledge error, [27-31](#)
 - architecture, [27-5](#)
 - auto-transmit mode, [27-16](#)
 - bit error, [27-30](#)
 - bit timing, [27-11](#)
 - block diagram, [27-4](#)
 - bus interface, [27-2](#)
 - clock, [27-11](#)
 - code examples, [27-91](#)
 - configuration mode, [27-10](#), [27-13](#)
 - CRC error, [27-31](#)
 - data field filtering, [27-20](#)
 - debug and test modes, [27-35](#)
 - disabling mailboxes, [27-23](#)
 - enabling mailboxes, [27-93](#)
 - error frames, [27-29](#), [27-31](#)
 - error levels, [27-33](#)
 - errors, [27-30](#)
 - event counter, [27-29](#)
 - extended frame, [27-10](#)
 - features, [27-2](#)
 - form error, [27-31](#)
 - global interrupts, [27-26](#)
 - hibernate state, [27-40](#)
 - identifier frame, [27-10](#)
 - initializing, [27-91](#)
 - initializing mailboxes, [27-93](#)
 - initiating transfers, [27-95](#)
 - interrupts, [27-25](#), [27-95](#)

- lost arbitration, [27-29](#)
- and low power designs, [27-40](#)
- low power features, [27-39](#)
- mailbox area registers, [27-6](#)
- mailbox control, [27-7](#)
- mailboxes, [27-5](#)
- mailbox interrupts, [27-25](#)
- mailbox RAM, [27-5](#)
- message buffers, [27-5](#)
- message received, [27-29](#)
- message stored, [27-30](#)
- multiplexing of signals, [27-3](#)
- nominal bit rate, [27-12](#)
- nominal bit time, [27-11](#)
- overload frame, [27-29](#)
- propagation segment, [27-12](#)
- protocol basics, [27-8](#)
- receive message lost, [27-29](#)
- receive message rejected, [27-29](#)
- receive operation, [27-16](#)
- receive operation flow chart, [27-19](#)
- registers, table, [27-41](#)
- remote frames, [27-22](#)
- re-synchronization, [27-12](#)
- retransmission, [27-14](#)
- sampling, [27-12](#)
- single shot transmission, [27-15](#)
- sleep mode, [27-40](#)
- software reset, [27-13](#)
- standard frame, [27-9](#)
- stuff error, [27-31](#)
- suspend mode, [27-39](#)
- test modes, [27-38](#)
- time quantum, [27-11](#)
- time stamps, [27-21](#)
- transceiver interconnection, [27-2](#)
- transmission, [27-9](#)
- transmission aborted, [27-29](#)
- transmission succeeded, [27-29](#)
- transmit operation, [27-13](#)
- transmit operation flow chart, [27-15](#)
- universal counter as event counter, [27-29](#)
- valid message, [27-30](#)
- wakeup from hibernate, [27-40](#)
- warnings, [27-30](#)
- watchdog mode, [27-21](#)
- CAN_AMxxH (acceptance mask register), [27-53](#)
- CAN_AMxxL (acceptance mask register), [27-42](#), [27-55](#)
- CAN_CEC (CAN error counter register), [27-37](#)
- Cancel Asynchronous Packet Transmission (CANCELAP) bit, [21-78](#)
- Cancel Control Message Transmission (CANCELCM) bit, [21-85](#)
- CAN_CLOCK (CAN clock register), [27-11](#)
- CAN controller abort acknowledge (CANx_AA1) register 1, [27-80](#)
- CAN controller abort acknowledge (CANx_AA2) register 2, [27-80](#)
- CAN controller acceptance mask (CANx_AMxxH) registers, [27-53](#)
- CAN controller acceptance mask (CANx_AMxxL) registers, [27-53](#)
- CAN controller clock (CANx_CLOCK) register, [27-48](#)
- CAN controller debug (CANx_DEBUG) register, [27-48](#)
- CAN controller error counter (CANx_CEC) register, [27-89](#)
- CAN controller error counter warning level (CANx_EWR) register, [27-90](#)
- CAN controller error status (CANx_ESR) register, [27-89](#)
- CAN controller global interrupt flag (CANx_GIF) register, [27-52](#)
- CAN controller global interrupt mask (CANx_GIM) register, [27-50](#)

Index

- CAN controller global interrupt status (CANx_GIS) register, [27-51](#)
- CAN controller global status (CANx_STATUS) register, [27-47](#)
- CAN controller interrupt pending (CANx_INTR) register, [27-49](#)
- CAN controller mailbox configuration (CANx_MC1) register 1, [27-73](#)
- CAN controller mailbox configuration (CANx_MC2) register 2, [27-73](#)
- CAN controller mailbox direction (CANx_MD1) register 1, [27-74](#)
- CAN controller mailbox direction (CANx_MD2) register 2, [27-74](#)
- CAN controller mailbox interrupt mask (CANx_MBIM1) register 1, [27-84](#)
- CAN controller mailbox interrupt mask (CANx_MBIM2) register 2, [27-84](#)
- CAN controller mailbox receive interrupt flag (CANx_MBRIF1) register 1, [27-86](#)
- CAN controller mailbox receive interrupt flag (CANx_MBRIF2) register 2, [27-86](#)
- CAN controller mailbox transmit interrupt flag (CANx_MBTIF1) register 1, [27-85](#)
- CAN controller mailbox transmit interrupt flag (CANx_MBTIF2) register 2, [27-85](#)
- CAN controller mailbox word 0 (CANx_MBxx_DATA0) register, [27-65](#)
- CAN controller mailbox word 1 (CANx_MBxx_DATA1) register, [27-65](#)
- CAN controller mailbox word 2 (CANx_MBxx_DATA2) register, [27-65](#)
- CAN controller mailbox word 3 (CANx_MBxx_DATA3) register, [27-65](#)
- CAN controller mailbox word 4 (CANx_MBxx_LENGTH) register, [27-63](#)
- CAN controller mailbox word 5 (CANx_MBxx_TIMESTAMP) register, [27-61](#)
- CAN controller mailbox word 6 (CANx_MBxx_ID0) register, [27-59](#)
- CAN controller mailbox word 7 (CANx_MBxx_ID1) register, [27-57](#)
- CAN controller master control (CANx_CONTROL) register, [27-46](#)
- CAN controller overwrite protection/single shot transmission (CANx_OPSS1) register 1, [27-77](#)
- CAN controller overwrite protection/single shot transmission (CANx_OPSS2) register 2, [27-77](#)
- CAN controller receive message lost (CANx_RML1) register 1, [27-76](#)
- CAN controller receive message lost (CANx_RML2) register 2, [27-76](#)
- CAN controller receive message pending (CANx_RMP1) register 1, [27-75](#)
- CAN controller receive message pending (CANx_RMP2) register 2, [27-75](#)
- CAN controller remote frame handling (CANx_RFH1) register 1, [27-83](#)
- CAN controller remote frame handling (CANx_RFH2) register 2, [27-83](#)
- CAN controller temporary mailbox disable feature (CANx_MBTD) register, [27-82](#)
- CAN controller timing (CANx_TIMING) register, [27-49](#)
- CAN controller transmission acknowledge (CANx_TA1) register 1, [27-81](#)

- CAN controller transmission acknowledge (CANx_TA2) register 2, [27-81](#)
- CAN controller transmission request reset (CANx_TRR1) register 1, [27-79](#)
- CAN controller transmission request reset (CANx_TRR2) register 2, [27-79](#)
- CAN controller transmission request set (CANx_TRS1) register 1, [27-78](#)
- CAN controller transmission request set (CANx_TRS2) register 2, [27-78](#)
- CAN controller universal counter (CANx_UCCNT) register, [27-88](#)
- CAN controller universal counter configuration mode (CANx_UCCNF) register, [27-87](#)
- CAN controller universal counter reload/capture (CANx_UCRC) register, [27-88](#)
- CAN_DEBUG (CAN debug register), [27-35](#), [27-36](#)
- CAN_EWR (CAN controller error counter warning level) register, [27-45](#)
- CAN_MBxx_DATA0 (mailbox word 0 register), [27-42](#)
- CAN_MBxx_DATA1 (mailbox word 1 register), [27-42](#)
- CAN_MBxx_DATA2 (mailbox word 2 register), [27-42](#)
- CAN_MBxx_DATA registers, [27-6](#)
- CAN_MBxx_ID0 (mailbox word 6 register), [27-6](#), [27-42](#)
- CAN_MBxx_ID1 (mailbox word 7 register), [27-6](#)
- CAN_MBxx_LENGTH (mailbox word 4 register), [27-6](#)
- CAN_MBxx_TIMESTAMP (mailbox word 5 register), [27-6](#)
- CAN_TIMING (CAN timing register), [27-11](#)
- CANx_AA1 (CAN controller abort acknowledge) register 1, [27-80](#)
- CANx_AA2 (CAN controller abort acknowledge) register 2, [27-80](#)
- CANx_AAx (CAN controller abort acknowledge) registers, [27-43](#)
- CANx_AMxxH (CAN controller acceptance mask) registers, [27-42](#), [27-53](#)
- CANx_AMxxL (CAN controller acceptance mask) registers, [27-42](#), [27-53](#)
- CANx_CEC (CAN controller error counter) register, [27-45](#), [27-89](#)
- CANx_CLOCK (CAN controller clock) register, [27-42](#), [27-48](#)
- CANx_CONTROL master control register, [27-41](#), [27-46](#)
- CANx_DEBUG CAN controller debug register, [27-41](#), [27-48](#)
- CANx_ESR (CAN controller error status) register, [27-45](#), [27-89](#)
- CANx_EWR (CAN controller error counter warning level) register, [27-90](#)
- CANx_GIF (CAN controller global interrupt flag) register, [27-42](#), [27-52](#)
- CANx_GIM (CAN controller global interrupt mask) register, [27-42](#), [27-50](#)
- CANx_GIS (CAN controller global interrupt status) register, [27-42](#), [27-51](#)
- CANx_INTR (CAN controller interrupt pending) register, [27-42](#), [27-49](#)
- CANx_MBIM1 (CAN controller mailbox interrupt mask) register 1, [27-84](#)
- CANx_MBIM2 (CAN controller mailbox interrupt mask) register 2, [27-84](#)
- CANx_MBIMx (CAN controller mailbox interrupt mask) registers, [27-44](#)
- CANx_MBRIF1 (CAN controller mailbox receive interrupt flag) register 1, [27-86](#)

Index

- CANx_MBRIF2 (CAN controller mailbox receive interrupt flag) register 2, [27-86](#)
- CANx_MBRIFx (CAN controller mailbox receive interrupt flag) registers, [27-44](#)
- CANx_MBTD (CAN controller temporary mailbox disable feature) register, [27-43](#), [27-82](#)
- CANx_MBTIF1 (CAN controller mailbox transmit interrupt flag) register 1, [27-85](#)
- CANx_MBTIF2 (CAN controller mailbox transmit interrupt flag) register 2, [27-85](#)
- CANx_MBTIFx (CAN controller mailbox transmit interrupt flag) registers, [27-44](#)
- CANx_MBxx_DATA0 (CAN controller mailbox word 0) register, [27-42](#), [27-65](#)
- CANx_MBxx_DATA1 (CAN controller mailbox word 1) register, [27-42](#), [27-65](#)
- CANx_MBxx_DATA2 (CAN controller mailbox word 2) register, [27-42](#), [27-65](#)
- CANx_MBxx_DATA3 (CAN controller mailbox word 3) register, [27-42](#), [27-65](#)
- CANx_MBxx_ID0 (CAN controller mailbox word 6) register, [27-42](#), [27-59](#)
- CANx_MBxx_ID1 (CAN controller mailbox word 7) register, [27-42](#), [27-57](#)
- CANx_MBxx_LENGTH (CAN controller mailbox word 4) register, [27-42](#), [27-63](#)
- CANx_MBxx_TIMESTAMP (CAN controller mailbox word 5) register, [27-42](#), [27-61](#)
- CANx_MC1 (CAN controller mailbox configuration) register 1, [27-73](#)
- CANx_MC2 (CAN controller mailbox configuration) register 2, [27-73](#)
- CANx_MCx (CAN controller mailbox configuration) registers, [27-43](#)
- CANx_MD1 (CAN controller mailbox direction) register 1, [27-74](#)
- CANx_MD2 (CAN controller mailbox direction) register 2, [27-74](#)
- CANx_MDx (CAN controller mailbox direction) registers, [27-43](#)
- CANx_OPSS1 (CAN controller overwrite protection/single shot transmission) register 1, [27-77](#)
- CANx_OPSS2 (CAN controller overwrite protection/single shot transmission) register 2, [27-77](#)
- CANx_OPSSx (CAN controller overwrite protection/single shot transmission) registers, [27-43](#)
- CANx_RFH1 (CAN controller remote frame handling) register 1, [27-83](#)
- CANx_RFH2 (CAN controller remote frame handling) register 2, [27-83](#)
- CANx_RFHx (CAN controller remote frame handling enable) registers, [27-44](#)
- CANx_RML1 (CAN controller receive message lost) register 1, [27-76](#)
- CANx_RML2 (CAN controller receive message lost) register 2, [27-76](#)
- CANx_RMLx (CAN controller receive message lost) registers, [27-43](#)
- CANx_RMP1 (CAN controller receive message pending) register 1, [27-75](#)
- CANx_RMP2 (CAN controller receive message pending) register 2, [27-75](#)
- CANx_RMPx (CAN controller receive message pending) registers, [27-43](#)

- CANxRX bit, [27-49](#)
- CANxRX pin, [27-8](#)
- CANx_STATUS (CAN controller global status) register, [27-41](#), [27-47](#)
- CANx_TA12 (CAN controller transmission acknowledge) register 2, [27-81](#)
- CANx_TA1 (CAN controller transmission acknowledge) register 1, [27-81](#)
- CANx_TAx (CAN controller transmit acknowledge) registers, [27-43](#)
- CANx_TIMING (CAN controller timing) register, [27-42](#), [27-49](#)
- CANx_TRR1 (CAN controller transmission request reset) register 1, [27-79](#)
- CANx_TRR2 (CAN controller transmission request reset) register 2, [27-79](#)
- CANx_TRRx (CAN controller transmit request reset) registers, [27-43](#)
- CANx_TRS1 (CAN controller transmission request set) register 1, [27-78](#)
- CANx_TRS2 (CAN controller transmission request set) register 2, [27-78](#)
- CANx_TRSx (CAN controller transmit request set) registers, [27-43](#)
- CANxTX bit, [27-49](#)
- CANxTX pin, [27-8](#)
- CANx_UCCNF (CAN controller universal counter configuration mode) register, [27-44](#), [27-87](#)
- CANx_UCCNT (CAN controller universal counter) register, [27-44](#), [27-88](#)
- CANx_UCRC (CAN controller universal counter reload/capture) register, [27-44](#), [27-88](#)
- CCA bit, [27-47](#)
- CCITT G.711 specification, [30-31](#)
- CCR bit, [27-46](#)
- CDE bit, [27-35](#), [27-48](#)
- CFIFO_ERR (chroma FIFO error) bit, [26-89](#)
- CFIFO_ERR (Chroma FIFO Overflow Error) bit, [24-50](#), [24-56](#), [24-57](#)
- Channel-In-Use (CIUx) bit, [21-58](#)
- channels
 - defined, serial, [30-24](#)
 - serial port TDM, [30-25](#)
 - serial select offset, [30-24](#)
- charge VBUS end interrupt enable (CHRG_VBUS_END_ENA) bit, [32-126](#)
- charge VBUS start interrupt enable (CHRG_VBUS_START_ENA) bit, [32-126](#)
- CHNL[9:0] field, [30-71](#), [30-72](#)
- CHRG_VBUS_END_ENA (charge VBUS end interrupt enable) bit, [32-126](#)
- CHRG_VBUS_START_ENA (charge VBUS start interrupt enable) bit, [32-126](#)
- chroma FIFO error (CFIFO_ERR) bit, [26-89](#)
- Chroma FIFO Overflow Error (CFIFO_ERR) bit, [24-50](#), [24-56](#), [24-57](#)
- circuit board testing, [B-1](#), [B-5](#)
- CLEAR_DATATOGGLE_R (reset endpoint data toggle) bit, [32-109](#)
- CLEAR_DATATOGGLE_T (reset endpoint data toggle) bit, [32-102](#)
- CLKDIV_BYPASS (bypass divisor) bit, [23-23](#)
- CLKDIV (clock divisor) bits, [32-133](#)
- CLKDIV (SD_CLK divisor) bits, [23-23](#)

Index

- CLK_E (SD_CLK bus clock enable) bit, [23-23](#)
- CLKHI[7:0] field, [29-37](#)
- CLKIN, [20-32](#)
- CLKLOW[7:0] field, [29-37](#)
- CLKS_EN (clocks enable) bit, [23-37](#)
- clock
 - clock signals, [20-32](#)
 - frequency for SPORT, [30-68](#)
 - SPI clock signal, [28-5](#)
- clock divide modulus register, [30-68](#)
- clock divisor (CLKDIV) bits, [32-133](#)
- clock phase, SPI, [28-17](#), [28-19](#)
- clock phase (CPHA) bit, [28-45](#)
- clock polarity, SPI, [28-17](#)
- clock polarity (CPOL) bit, [28-45](#)
- clock rate
 - SPORT, [30-2](#)
- clocks enable (CLKS_EN) bit, [23-37](#)
- CM Allocate Channel List field, [21-144](#)
- CM Allocate Number Free field, [21-143](#)
- CM Allocate Number Requested field, [21-141](#)
- CM Allocate Status, [21-143](#)
- CM Allocate Status Encodings, [21-143](#)
- CMD_ACT (CMD active) bit, [23-30](#)
- CMD active (CMD_ACT) bit, [23-30](#)
- CMD active mask (CMD_ACT_MASK) bit, [23-33](#)
- CMD_ACT_MASK (CMD active mask) bit, [23-33](#)
- CM Data field, [21-135](#)
- CMD_CRC_FAIL (CMD CRC fail) bit, [23-30](#)
- CMD_CRC_FAIL_MASK bit, [23-33](#)
- CMD_CRC_FAIL_STAT bit, [23-32](#)
- CM De-Allocate Connection Label field, [21-145](#)
- CM De-Allocate Status, [21-147](#)
- CM De-Allocate Status Encodings, [21-147](#)
- CMD_E (command enable) bit, [23-24](#)
- CM Destination Address, [21-132](#)
- CMD_IDX (command index) bits, [23-24](#)
- CMD_INT_E (command interrupt) bit, [23-24](#)
- CMD_L_RSP (long response) bit, [23-24](#)
- CMD_PEND_E (command pending) bit, [23-24](#)
- CMD_RESP_END (CMD response end) bit, [23-30](#)
- CMD_RESP_END_MASK (CMD response end mask) bit, [23-33](#)
- CMD_RESP_END_STAT (CMD response end status) bit, [23-32](#)
- CMD response end (CMD_RESP_END) bit, [23-30](#)
- CMD response end mask (CMD_RESP_END_MASK) bit, [23-33](#)
- CMD response end status (CMD_RESP_END_STAT) bit, [23-32](#)
- CMD_RSP (response) bit, [23-24](#)
- CMD_SENT bit, [23-30](#)
- CMD_SENT_MASK bit, [23-33](#)
- CMD_SENT_STAT bit, [23-32](#)
- CMD_TIMEOUT (CMD time out) bit, [23-30](#)
- CMD_TIMEOUT_MASK bit, [23-33](#)
- CMD_TIMEOUT_STAT, [23-32](#)
- CM GetSource Channel field, [21-148](#)
- CM Message Type Encodings, [21-133](#)
- CM Message Type field, [21-133](#)
- CM Priority, [21-132](#)
- CM Read Address field, [21-137](#)
- CMREN, [21-46](#)
- CM Source Address, [21-132](#)
- CMTCEN, [21-46](#)
- CM Transmission Status, [21-134](#)
- CM Transmission Status field, [21-133](#)

- CMTSEN, [21-46](#)
- CM Write Address field, [21-139](#)
- CM Write Data field, [21-139](#)
- CM Write Length field, [21-15](#)
- CNOS (tuning of DPHY clocks) bits, [32-130](#)
- codecs, connecting to, [30-2](#)
- COLDRV_SCALE (column driver scale value) bits, [22-16](#)
- column driver scale value (COLDRV_SCALE) bits, [22-16](#)
- column enable width (KPAD_COLEN) bits, [22-10](#)
- columns value pressed (KPAD_COL) bits, [22-16](#)
- command enable (CMD_E) bit, [23-24](#)
- command index (CMD_IDX) bits, [23-24](#)
- command interrupt (CMD_INT_E) bit, [23-24](#)
- command pending (CMD_PEND_E) bit, [23-24](#)
- commands
 - transfer initiate, [28-27](#)
- companding, [30-17](#), [30-26](#)
 - defined, [30-31](#)
 - lengths supported, [30-31](#)
 - multichannel operations, [30-26](#)
- configuration
 - CAN, [27-13](#)
 - SPORT, [30-12](#)
- CONN_B (connection indicator) bit, [32-89](#)
- CONN_BE (connection IRQ enable) bit, [32-90](#)
- connection indicator (CONN_B) bit, [32-89](#)
- connection IRQ enable (CONN_BE) bit, [32-90](#)
- Connection Label, [21-144](#)
- Connection Label (CLx) field, [21-57](#)
- Control Message Arbitrating (CMARB) bit, [21-23](#)
- Control Message Interrupt, [21-30](#)
- Control Message Receive Buffer, [21-15](#)
- Control Message Receive Buffer (CMRB), [21-84](#)
- Control Message Receive Buffer Entry Field Offsets, [21-152](#)
- Control Message Receive Buffer Entry Offsets, [21-152](#)
- Control Message Receive Buffer Entry x (CMRBEx) bits, [21-85](#)
- Control Message Receive Buffer Overflow (CMROF) interrupt event, [21-37](#)
- Control Message Receive Buffer Overflow interrupt enable, [21-46](#)
- Control Message Received (CMR) bit, [21-24](#)
- Control Message Received (CMR) interrupt event, [21-37](#)
- Control Message Received interrupt enable, [21-46](#)
- Control Message Reception, [21-150](#)
- Control Message Transmission, [21-131](#)
- Control Message Transmit Buffer Busy (CMBSY) bit, [21-23](#)
- Control Message Transmit Buffer (CMTB), [21-83](#)
- Control Message Transmit Buffer Successfully Cancelled (CMTTC) interrupt event, [21-38](#)
- Control Message Transmit Buffer Successfully Cancelled interrupt enable, [21-46](#)
- Control Message Transmit Buffer Successfully Sent (CMTS) interrupt event, [21-38](#)
- Control Message Transmit Buffer Successfully Sent interrupt enable, [21-46](#)

Index

Control Message Transmitting (CMTX) bit, [21-24](#)
conventions, [-xlviii](#)
core clock, *See* CCLK
Count Position (COUNTPOSx) field, [21-68](#)
CPHA bit, [28-45](#)
CPHA (clock phase) bit, [28-45](#)
CPOL bit, [28-45](#)
CPOL (clock polarity) bit, [28-45](#)
CRCE bit, [27-89](#)
CRC error, [21-136](#)
CrossCore software, [20-36](#)
crystal oscillator pins, [21-3](#)
CSA bit, [27-39](#), [27-47](#)
CSR bit, [27-39](#), [27-46](#)
CSR_HBR (USB hibernate signal) bit, [32-130](#)
CSR_RSTD (USB pu/pd restore control) bit, [32-130](#)
CTS (clear to send) bit, [31-39](#)
customer support, [-xlv](#)

D

data block end (DAT_BLK_END) bit, [23-30](#)
data block end mask (DAT_BLK_END_MASK) bit, [23-33](#)
data block end status (DAT_BLK_END_STAT) bit, [23-32](#)
data corruption, avoiding with SPI, [28-19](#)
DATA_COUNT (data count) bits, [23-29](#)
data count (DATA_COUNT) bits, [23-29](#)
DATAEND (data end indicator) bit, [32-98](#)
data end (DAT_END) bit, [23-30](#)
data end indicator (DATAEND) bit, [32-98](#)
DATAERROR_R (load error indicator) bit, [32-109](#)
data field byte 0[7:0] field, [27-65](#)

data field byte 1[7:0] field, [27-65](#)
data field byte 2[7:0] field, [27-67](#)
data field byte 3[7:0] field, [27-67](#)
data field byte 4[7:0] field, [27-69](#)
data field byte 5[7:0] field, [27-69](#)
data field byte 6[7:0] field, [27-71](#)
data field byte 7[7:0] field, [27-71](#)
data field filtering, CAN, [27-20](#)
data formats, SPORT, [30-30](#)
DATA_LENGTH (data length) bits, [23-27](#)
data length (DATA_LENGTH) bits, [23-27](#)
data move, serial port operations, [30-40](#)
data packet in FIFO indicator (FIFO_NOT_EMPTY_T) bit, [32-102](#)
data packet in FIFO indicator (RXPKTRDY_R) bit, [32-109](#)
data packet in FIFO indicator (TXPKTRDY) bit, [32-98](#)
data packet in FIFO indicator (TXPKTRDY_T) bit, [32-102](#)
data packet receive indicator (RXPKTRDY) bit, [32-98](#)
data sampling, serial, [30-35](#)
data transfer block length (DTX_BLK_LGTH) bits, [23-27](#)
data transfer direction (DTX_DIR) bit, [23-27](#)
data transfer DMA enable (DTX_DMA_E) bit, [23-27](#)
data transfer enable (DTX_E) bit, [23-27](#)
data transfer mode (DTX_MODE) bit, [23-27](#)
data transfers
SPI, [28-20](#)
data word, serial data formats, [30-61](#)
DAT_BLK_END (data block end) bit, [23-30](#)

- DAT_BLK_END_MASK (data block end mask) bit, [23-33](#)
- DAT_BLK_END_STAT (data block end status) bit, [23-32](#)
- DAT_CRC_FAIL (data CRC fail) bit, [23-30](#)
- DAT_CRC_FAIL_MASK bit, [23-33](#)
- DAT_CRC_FAIL_STAT bit, [23-32](#)
- DAT_END (data end) bit, [23-30](#)
- DAT_END_MASK bit, [23-33](#)
- DAT_END_STAT bit, [23-32](#)
- DAT_TIMEOUT (data time out) bit, [23-30](#)
- DAT_TIMEOUT_MASK bit, [23-33](#)
- DAT_TIMEOUT_STAT bit, [23-32](#)
- DBON_SCALE (debounce scale value) bits, [22-16](#)
- DCNT[7:0] field, [29-41](#), [29-42](#)
- DDR SDRAM controller, [20-14](#)
- debounce scale value (DBON_SCALE) bits, [22-16](#)
- debugging, [20-37](#)
- DEC bit, [27-37](#), [27-48](#)
- deep sleep mode, [20-34](#)
- Delay Register Updated (DRU) interrupt event, [21-32](#)
- Delay Register Updated interrupt enable, [21-45](#)
- DERREN, [21-46](#)
- detected FIFO not empty (FIFO_FULL_R) bit, [32-109](#)
- DEV_ADDR (device address) bit, [24-53](#)
- development tools, [20-36](#)
- device address (DEV_ADDR) bit, [24-53](#)
- device DASP to host line status (ATAPI_DASP) bit, [24-60](#)
- device interrupt mask (ATAPI_DEV_INT_MASK) bit, [24-56](#)
- device interrupt status (ATAPI_DEV_INT) bit, [24-58](#)
- device interrupt to host line status (ATAPI_INTR) bit, [24-60](#)
- device receive buffer (REG_RXBUFFER) bits, [24-55](#)
- device terminate multi-DMA transfer interrupt mask (MULTI_TERM_MASK) bit, [24-56](#)
- device terminate multi-DMA transfer interrupt status (MULTI_TERM_INT) bit, [24-58](#)
- device terminate ultra-DMA-in transfer interrupt mask (UDMAIN_TERM_MASK) bit, [24-56](#)
- device terminate ultra-DMA-in transfer interrupt status (UDMAIN_TERM_INT) bit, [24-58](#)
- device terminate ultra-DMA-out transfer interrupt mask (UDMAOUT_TERM_MASK) bit, [24-56](#)
- device terminate ultra-DMA-out transfer interrupt status (UDMAOUT_TERM_INT) bit, [24-58](#)
- device transmit buffer (REG_TXBUFFER) bits, [24-54](#)
- DEV_RST (device reset) bit, [24-50](#)
- DFC[15:0] field, [27-59](#)
- DF (divide CLKIN by 2) bit, [32-132](#)
- DFM[15:0] field, [27-55](#)
- DIL bit, [27-37](#), [27-48](#)
- DIOR/DIOW asserted pulsewidth (TD) bits, [24-66](#)
- DIOR/DIOW pulsewidth (T2_REG_PIO) bits, [24-65](#)
- DIOW data hold (T4_REG) bits, [24-65](#)
- DIR (direction) bit, [26-82](#)

Index

- DIRECTION (DMA Tx or Rx selection)
 - bit, [32-135](#)
- direct memory access, *See* DMA
- disable nyet handshake (DISNYET) bit,
 - [32-109](#)
- discharge VBUS end interrupt enable (DISCHRG_VBUS_END_ENA)
 - bit, [32-126](#)
- discharge VBUS start interrupt enable (DISCHRG_VBUS_START_ENA)
 - bit, [32-126](#)
- DISCHRG_VBUS_END_ENA
 - (discharge VBUS end interrupt enable) bit, [32-126](#)
- DISCHRG_VBUS_START_ENA
 - (discharge VBUS start interrupt enable) bit, [32-126](#)
- DISCON_B (disconnect/session end indicator) bit, [32-89](#)
- DISCON_BE (disconnect/session end IRQ enable) bit, [32-90](#)
- disconnect/session end indicator (DISCON_B) bit, [32-89](#)
- disconnect/session end IRQ enable (DISCON_BE) bit, [32-90](#)
- DISNYET (disable nyet handshake) bit,
 - [32-109](#)
- DITFS bit, [30-39](#), [30-51](#), [30-55](#), [30-66](#)
- divide CLKIN by 2 (DF) bit, [32-132](#)
- divisor latch high byte[15:8] field, [31-48](#)
- divisor latch low byte[7:0] field, [31-48](#)
- divisor reset, UART, [31-49](#)
- DLC[3:0] field, [27-63](#)
- DLEN (data length) bits, [26-83](#)
- DMA
 - buffer size, multichannel SPORT, [30-26](#)
 - for SPI transmit, [28-14](#)
 - overview, [20-11](#)
 - serial port block transfers, [30-40](#)
 - and SPI, [28-14](#)
 - SPI data transmission, [28-15](#), [28-16](#)
 - SPI master, [28-33](#)
 - SPI slave, [28-35](#)
 - and SPORT, [30-3](#)
 - support for peripherals, [20-4](#)
 - and UART, [31-25](#), [31-44](#)
- DMA0 Urgent Request (DMA0URQ) bit,
 - [24-50](#), [24-56](#), [24-57](#)
- DMA0URQ (DMA0 Urgent Request) bit,
 - [24-50](#), [24-56](#), [24-57](#)
- DMA0URQ (DMA0 urgent request) bit,
 - [26-89](#)
- DMA1 Urgent Request (DMA1URQ) bit,
 - [24-50](#), [24-56](#), [24-57](#)
- DMA1URQ (DMA1 Urgent Request) bit,
 - [24-50](#), [24-56](#), [24-57](#)
- DMA1URQ (DMA1 urgent request) bit,
 - [26-89](#)
- DMAACTIVEx bits, [21-28](#)
- DMA address high (DMA_ADDR_HIGH) bits, [32-138](#)
- DMA address low (DMA_ADDR_LOW) bits, [32-137](#)
- DMA_ADDR_HIGH (DMA address high) bits, [32-138](#)
- DMA_ADDR_LOW (DMA address low) bits, [32-137](#)
- DMA burst mode selection (BURST_MODE) bits, [32-135](#)
- DMA bus error (BUSERROR) bit, [32-135](#)
- DMACFG (one/two DMA channel modes) bit, [26-83](#)
- DMA Channel x Done interrupt enable, [21-48](#)

- DMA Channel x Half Done interrupt enable, [21-48](#)
- DMA_COUNT_LOW (DMA count low) bits, [32-139](#), [32-140](#)
- DMA count low (DMA_COUNT_LOW) bits, [32-139](#), [32-140](#)
- DMA_CSTATE (DMA mode state machine current state) bits, [24-61](#)
- DMA enable (DMA_ENA) bit, [32-135](#)
- DMA_ENA (DMA enable) bit, [32-135](#)
- DMA Error Channel Number (DERRNUM) field, [21-26](#)
- DMA Error (DERR) interrupt event, [21-39](#)
- DMA Error interrupt enable, [21-46](#)
- DMA mode 0/1 selection (MODE) bit, [32-135](#)
- DMA mode select (DMAREQMODE_R) bit, [32-109](#)
- DMA mode select (DMAREQMODE_RH) bit, [32-109](#)
- DMA mode select (DMAREQMODE_T) bit, [32-102](#)
- DMA mode state machine current state (DMA_CSTATE) bits, [24-61](#)
- DMAPMEN_x, [21-28](#)
- DMAREQ_ENA_R (DMA request enable) bit, [32-109](#)
- DMAREQ_ENA_T (DMA request enable) bit, [32-102](#)
- DMAREQMODE_R (DMA mode select) bit, [32-109](#)
- DMAREQMODE_RH (DMA mode select) bit, [32-109](#)
- DMAREQMODE_T (DMA mode select) bit, [32-102](#)
- DMA request enable (DMAREQ_ENA_R) bit, [32-109](#)
- DMA request enable (DMAREQ_ENA_T) bit, [32-102](#)
- DMA support, [20-11](#)
 - DMAC0, [20-11](#)
 - DMAC1, [20-11](#)
- DMA Tx or Rx selection (DIRECTION) bit, [32-135](#)
- DMAx Bit-Swap Enable (BITSWAPEN_x) bit, [21-62](#)
- DMAx Direction (DD_x) bit, [21-61](#)
- DMAx Done (DONEx) interrupt event, [21-41](#)
- DMAx Half-Done (HDONEx) interrupt event, [21-41](#)
- DMAx_INT (USB DMA endpoint x interrupt) bits, [32-134](#)
- DMAx Logical Channel (LCHAN_x) field, [21-61](#)
- DMAx Operation Flow (MFLOW_x) field, [21-63](#)
- DNACK bit, [29-15](#), [29-45](#)
- DNM bit, [27-46](#)
- DONEEN_x, [21-48](#)
- DR bit, [31-18](#)
- DR (data ready) bit, [31-36](#), [31-37](#)
- DR flag, [31-24](#)
- DRI bit, [27-37](#), [27-48](#)
- DRIVE_VBUS_OFF_ENA (drive VBUS off interrupt enable) bit, [32-126](#)
- drive VBUS off interrupt enable (DRIVE_VBUS_OFF_ENA) bit, [32-126](#)
- DRIVE_VBUS_ON_ENA (drive VBUS on interrupt enable) bit, [32-126](#)
- drive VBUS on interrupt enable (DRIVE_VBUS_ON_ENA) bit, [32-126](#)
- DRUEN, [21-45](#)
- DRxPRI signal, [30-5](#)
- DRxPRI SPORT input, [30-7](#)
- DRxSEC signal, [30-5](#)
- DRxSEC SPORT input, [30-7](#)

Index

- DTO bit, [27-37](#), [27-48](#)
- DTX_BLK_LGTH (data transfer block length) bits, [23-27](#)
- DTX_DIR (data transfer direction) bit, [23-27](#)
- DTX_DMA_E (data transfer DMA enable) bit, [23-27](#)
- DTX_E (data transfer enable) bit, [23-27](#)
- DTX_MODE (data transfer mode) bit, [23-27](#)
- DTxPRI signal, [30-5](#)
- DTxPRI SPORT output, [30-7](#)
- DTxSEC signal, [30-5](#)
- DTxSEC SPORT output, [30-7](#)
- dynamic power management, [20-32](#)

- E**
- early frame sync, [30-37](#)
- EBIU, [20-14](#)
- EBO bit, [27-47](#)
- ECC0 (parity calculation result0) bits, [25-24](#)
- ECC1 (parity calculation result1) bits, [25-24](#)
- ECC2 (parity calculation result2) bits, [25-24](#)
- ECC3 (parity calculation result3) bits, [25-24](#)
- ECCCNT (transfer count) bits, [25-25](#)
- ECC_RST (ECC (and NFC counters) reset bit, [25-25](#)
- ELSI bit, [31-10](#), [31-44](#), [31-45](#), [31-46](#)
- EMISO bit, [28-21](#), [28-45](#)
- EMISO (enable MISO) bit, [28-45](#)
- enable MISO (EMISO) bit, [28-45](#)
- ENABLE_SUSPENDM (suspend mode output enable) bit, [32-82](#)
- end of cycle time for PIO access transfers (TEOC_REG_PIO) bits, [24-65](#)
- end of cycle time for register access transfers (T2_REG) bits, [24-64](#)
- END_ON_TERM (end/terminate select) bit, [24-50](#)
- endpoint number (EPNUM) bits, [32-135](#)
- endpoint x Rx enable (EPx_RX_ENA) bits, [32-95](#)
- endpoint x Tx enable (EPx_TX_ENA) bits, [32-95](#)
- end/terminate select (END_ON_TERM) bit, [24-50](#)
- EN (enable) bit, [26-82](#)
- EP0_NAK_LIMIT (EP0 NAK limit) bits, [32-117](#)
- EP0 NAK limit (EP0_NAK_LIMIT) bits, [32-117](#)
- EP0_RX_COUNT (received byte count in EP0 FIFO) bits, [32-115](#)
- EP bit, [27-47](#)
- EP halted after a NAK (NAK_TIMEOUT_H) bit, [32-98](#), [32-102](#)
- EPIF bit, [27-28](#), [27-52](#)
- EPIM bit, [27-28](#), [27-50](#)
- EPIS bit, [27-28](#), [27-51](#)
- EPNUM (endpoint number) bits, [32-135](#)
- EPPI
 - block diagram, [26-5](#)
 - control signal polarities, [26-79](#)
 - data width, [26-79](#)
 - GP output, [26-13](#), [26-14](#)
 - operating modes, [26-79](#)
- EPPI clipping (EPPIx_CLIP) register, [26-77](#), [26-101](#)
- EPPI_CONTROL (EPPI control register), [26-79](#)
- EPPI control register (EPPI_CONTROL), [26-79](#)
- EPPI vertical transfer count (EPPIx_VCOUNT) register, [26-77](#)

- EPPIx_CLIP (EPPI clipping) register, [26-77](#), [26-101](#)
- EPPIx_CLKDIV (EPPIx clock divide) register, [26-77](#), [26-96](#)
- EPPIx clock divide (EPPIx_CLKDIV) register, [26-77](#), [26-96](#)
- EPPIx_CONTROL register, [26-77](#), [26-83](#)
- EPPIx_FRAME (EPPIx lines per frame) register, [26-77](#), [26-92](#)
- EPPIx_FS1P_AVPL (EPPIx FS1 period) register, [26-77](#), [26-99](#)
- EPPIx FS1 period (EPPIx_FS1P_AVPL) register, [26-77](#), [26-99](#)
- EPPIx_FS1W_HBL (EPPIx FS1 width) register, [26-77](#), [26-96](#)
- EPPIx FS1 width (EPPIx_FS1W_HBL) register, [26-77](#), [26-96](#)
- EPPIx FS2 period (EPPIx_FS2P_LAVF) register, [26-77](#), [26-100](#)
- EPPIx_FS2P_LAVF (EPPIx FS2 period) register, [26-77](#), [26-100](#)
- EPPIx FS2 width (EPPIx_FS2W_LVB) register, [26-77](#), [26-98](#)
- EPPIx_FS2W_LVB (EPPIx FS2 width) register, [26-77](#), [26-98](#)
- EPPIx_HCOUNT (EPPIx horizontal transfer count) register, [26-76](#), [26-95](#)
- EPPIx_HDELAY (EPPIx horizontal delay count) register, [26-77](#), [26-94](#)
- EPPIx horizontal delay count (EPPIx_HDELAY) register, [26-77](#), [26-94](#)
- EPPIx horizontal transfer count (EPPIx_HCOUNT) register, [26-76](#), [26-95](#)
- EPPIx_LINE (EPPIx samples per line) register, [26-77](#), [26-92](#)
- EPPIx lines per frame (EPPIx_FRAME) register, [26-77](#), [26-92](#)
- EPPIx samples per line (EPPIx_LINE) register, [26-77](#), [26-92](#)
- EPPIx_STATUS (EPPI status) register, [26-76](#), [26-89](#)
- EPPIx status (EPPIx_STATUS) register, [26-76](#), [26-89](#)
- EPPIx_VCOUNT (EPPI vertical transfer count) register, [26-77](#), [26-94](#)
- EPPIx_VDELAY (EPPI vertical delay count) register, [26-77](#), [26-93](#)
- EPPIx vertical delay count (EPPIx_VDELAY) register, [26-77](#), [26-93](#)
- EPPIx vertical transfer count (EPPIx_VCOUNT) register, [26-94](#)
- EPS (even parity select) bit, [31-30](#)
- EPx_RX_ENA (endpoint x Rx enable) bits, [32-95](#)
- EPx_RX_E (USB Rx endpoint x interrupt enable) bits, [32-88](#)
- EPx_RX (USB Rx endpoint x interrupt) bits, [32-86](#)
- EPx_TX_ENA (endpoint x Tx enable) bits, [32-95](#)
- EPx_TX_E (USB Tx endpoint x interrupt enable) bits, [32-87](#)
- EPx_TX (USB Tx endpoint x interrupt) bits, [32-85](#)
- ERBFI bit, [31-9](#), [31-18](#), [31-43](#), [31-44](#), [31-45](#)
- ERR_DET (Error Detected in Preamble) bit, [24-50](#), [24-56](#), [24-57](#)
- ERR_DET (preamble error detected) bit, [26-89](#)
- ERR_NCOR (Error Not Corrected in Preamble) bit, [24-50](#), [24-56](#), [24-57](#)
- ERR_NCOR (preamble error not corrected) bit, [26-89](#)

Index

- error
 - bus exception, [21-5](#)
 - hardware interrupt, [21-5](#)
- Error Detected in Preamble (ERR_DET)
 - bit, [24-50](#), [24-56](#), [24-57](#)
- error frames, CAN, [27-31](#)
- ERROR_H (timeout error) bit, [32-98](#)
- Error Not Corrected in Preamble (ERR_NCOR) bit, [24-50](#), [24-56](#), [24-57](#)
- error-passive interrupt, CAN, [27-28](#)
- ERROR_RH (timeout error indicator) bit, [32-109](#)
- error signals, SPI, [28-23](#) to [28-25](#)
- ERROR_TH (timeout error indicator) bit, [32-102](#)
- error warning receive interrupt, CAN, [27-28](#)
- error warning transmit interrupt, CAN, [27-28](#)
- ETBEI bit, [31-8](#), [31-17](#), [31-43](#), [31-44](#), [31-45](#)
- event counter, CAN, [27-29](#)
- EWLREC[7:0] field, [27-90](#)
- EWLTEC[7:0] field, [27-90](#)
- EWRIF bit, [27-28](#), [27-52](#)
- EWRIM bit, [27-28](#), [27-50](#)
- EWRIS bit, [27-28](#), [27-51](#)
- EWTIF bit, [27-28](#), [27-52](#)
- EWTIM bit, [27-28](#), [27-50](#)
- EWTIS bit, [27-28](#), [27-51](#)
- External Bus Interface Unit, [20-14](#)
- external crystal, [20-32](#)
- EXTTEST instruction, [B-5](#)
- EXTID[15:0] field, [27-55](#), [27-59](#)
- EXTID[17:16] field, [27-53](#), [27-57](#)
- EZ-KIT Lite, [20-39](#)

- F**
- F1_ACT bits, [26-100](#)
- F1VB_AD bits, [26-98](#)
- F1VB_BD bits, [26-98](#)
- F2_ACT bits, [26-100](#)
- F2VB_AD bits, [26-98](#)
- F2VB_BD bits, [26-98](#)
- FAST bit, [29-41](#), [29-43](#)
- fast mode, TWI, [29-11](#)
- FCPOL (flow control pin polarity) bit, [31-33](#)
- FCZ0EN, [21-45](#)
- FCZ1EN, [21-45](#), [21-46](#)
- FDf bit, [27-20](#), [27-53](#)
- FE (framing error) bit, [31-36](#), [31-37](#)
- FER bit, [27-89](#)
- FERREN, [21-45](#)
- FFE bit, [31-50](#), [31-51](#)
- Field (FLD) bit, [24-50](#), [24-56](#), [24-57](#)
- field (FLD) bit, [26-89](#)
- field select/trigger (FLD_SEL) bit, [26-82](#)
- FIFO_COUNT bits, [23-34](#)
- FIFO Error (FERR) interrupt event, [21-36](#)
- FIFO Error interrupt enable, [21-45](#)
- FIFO_FLUSH (flush FIFOs) bit, [24-50](#)
- FIFO_FULL_R (detected FIFO not empty) bit, [32-109](#)
- FIFO_NOT_EMPTY_T (data packet in FIFO indicator) bit, [32-102](#)
- FIFO regular watermark (FIFO_RWM) bits, [26-83](#)
- FIFO_RWM (FIFO regular watermark) bits, [26-83](#)
- FIFO urgent watermarks (FIFO_UWM) bits, [26-83](#)
- FIFO_UWM (FIFO urgent watermarks) bits, [26-83](#)
- Fixed Pattern Matching select (FIXEDPM) bit, [21-67](#)
- FLD (Field) bit, [24-50](#), [24-56](#), [24-57](#)
- FLD (field) bit, [26-89](#)
- FLD_SEL (field select/trigger) bit, [26-82](#)

- FLGx bit, [28-47](#)
- flow charts
 - CAN receive operation, [27-19](#)
 - CAN transmit operation, [27-15](#)
 - SPI core-driven, [28-39](#)
 - SPI DMA, [28-40](#)
 - TWI master mode, [29-33](#)
 - TWI slave mode, [29-32](#)
- FLSx bit, [28-12](#), [28-46](#)
- FLSx (slave select enable) bits, [28-46](#)
- flush endpoint FIFO (FLUSHFIFO) bit, [32-98](#)
- flush endpoint FIFO (FLUSHFIFO_R) bit, [32-109](#)
- flush endpoint FIFO (FLUSHFIFO_T) bit, [32-102](#)
- FLUSHFIFO (flush endpoint FIFO) bit, [32-98](#)
- FLUSHFIFO_R (flush endpoint FIFO) bit, [32-109](#)
- FLUSHFIFO_T (flush endpoint FIFO) bit, [32-102](#)
- FMD bit, [27-53](#)
- FORCE_DATATOGGLE_T (force endpoint data toggle) bit, [32-102](#)
- force endpoint data toggle (FORCE_DATATOGGLE_T) bit, [32-102](#)
- FORCE_MSEL (force PLL frequency multiplier) bits, [32-132](#)
- force PLL frequency multiplier (FORCE_MSEL) bits, [32-132](#)
- formatting enable (RGB_FMT_EN) bit, [26-83](#)
- FPE bit, [31-50](#), [31-51](#)
- Frame Counter 0, [A-12](#)
- Frame Counter 0 Zero (FCZ0) interrupt event, [21-34](#)
- Frame Counter 0 Zero interrupt enable, [21-45](#)
- Frame Counter 1, [A-12](#)
- Frame Counter 1 Zero (FCZ1) interrupt event, [21-34](#)
- Frame Counter 1 Zero interrupt enable, [21-45](#)
- framed serial transfers, characteristics, [30-33](#)
- framed/unframed data, [30-32](#)
- Frame Locked (FLOCK) bit, [21-25](#)
- Frame Locked to Unlocked (FL2U) interrupt event, [21-35](#)
- FRAME_NUMBER (USB frame number) bits, [32-91](#)
- frame sync
 - active high/low, [30-35](#)
 - early, [30-37](#)
 - early/late, [30-37](#)
 - external/internal, [30-34](#)
 - internal, [30-28](#)
 - internally generated, [30-69](#)
 - late, [30-37](#)
 - multichannel mode, [30-20](#)
 - sampling edge, [30-35](#)
 - SPORT options, [30-32](#)
- frame sync configuration (FS_CFG) bits, [26-82](#)
- frame sync divider[15:0] field, [30-69](#)
- frame synchronization and SPORT, [30-3](#)
- frame sync pulse
 - use of, [30-54](#)
 - when issued, [30-55](#)
- frame sync signal, control of, [30-54](#), [30-59](#)
- Frame Track Overflow Error (FTERR_OVR) bit, [24-50](#), [24-56](#), [24-57](#)
- frame track overflow (FTERR_OVR) bit, [26-89](#)

Index

Frame Track Underflow Error
(FTERR_UNDR) bit, 24-50, 24-56, 24-57

frame track underflow (FTERR_UNDR) bit, 26-89

Frame Unlocked to Locked (FU2L) interrupt event, 21-35

frequencies, clock and frame sync, 30-28

FS_CFG (frame sync configuration) bits, 26-82

FSDEV (full- or high-speed device indicator) bit, 32-122, 32-124

FSDR bit, 30-24, 30-71

FS_EOF1 (full-speed EOF 1) bits, 32-128

FTERR_OVR (frame track overflow) bit, 26-89

FTERR_OVR (Frame Track Overflow Error) bit, 24-50, 24-56, 24-57

FTERR_UNDR (frame track underflow) bit, 26-89

FTERR_UNDR (Frame Track Underflow Error) bit, 24-50, 24-56, 24-57

full duplex, 30-4, 30-7

SPI, 28-1

full on mode, 20-33

full- or high-speed device indicator (FSDEV) bit, 32-122, 32-124

full-speed EOF 1 (FS_EOF1) bits, 32-128

FUNCTION_ADDRESS (USB peripheral device address) bits, 32-81

G

GCALL bit, 29-16, 29-40

GEN bit, 29-37

general call address, TWI, 29-10

general-purpose ports, 20-15

get more data (GM) bit, 28-45

GIRQ bit, 27-49

glitch filtering, UART, 31-15

GLOBAL_ENA (USB enable) bit, 32-95

global interrupts, CAN, 27-26

GM bit, 28-29, 28-45

GM (get more data) bit, 28-45

GPIO, 21-3

Group cast/Broadcast Transmission Status Encodings, 21-134

H

H.100, 30-24, 30-27

HDONEENx, 21-48

hibernate state, 20-34

and CAN, 27-40

HIGH_EVEN (upper limit for even bytes (luma) bits, 26-101

HIGH_ODD (upper limit for odd bytes (chroma) bits, 26-101

high- or full-speed device indicator (FSDEV) bit, 32-122, 32-124

high-speed EOF 1 (HS_EOF1) bits, 32-128

high speed mode enable (HS_ENABLE) bit, 32-82

high speed mode flag (HS_MODE) bit, 32-82

HMVIP, 30-27

Host DMA interface, 20-13

host DMA port (HOSTDP), 20-13

host negotiation request (HOST_REQ) bit, 32-122, 32-124

HOST_REQ (host negotiation request) bit, 32-122, 32-124

host terminate current transfer interrupt mask (HOST_TERM_XFER_MASK) bit, 24-56

host terminate current transfer interrupt status (HOST_TERM_XFER_INT) bit, 24-58

host termination (ATAPI_HOST_TERM) bit, 24-61

- HOST_TERM_XFER_INT (host terminate current transfer interrupt status) bit, [24-58](#)
- HOST_TERM_XFER_MASK (host terminate current transfer interrupt mask) bit, [24-56](#)
- HS_ENABLE (high speed mode enable) bit, [32-82](#)
- HS_EOF1 (high-speed EOF 1) bits, [32-128](#)
- HS_MODE (high speed mode flag) bit, [32-82](#)
- ## I
- I²C bus standard, [20-16](#), [29-1](#)
- I²S, [20-20](#)
- format, [30-13](#)
 - serial devices, [30-2](#)
- ICLKGEN (internal clock generation) bit, [26-82](#)
- IDE bit, [27-57](#)
- IDE interface, [24-1](#)
- IEEE 1149.1 standard, *See* JTAG standard
- IFSGEN (internal frame sync generation) bit, [26-82](#)
- INCOMPTX_RH (large packet split) bit, [32-109](#)
- INCOMPTX_R (large packet split) bit, [32-109](#)
- INCOMPTX_T (large packet split) bit, [32-102](#)
- increase PLL charge pump current (TM_SEL) bit, [32-132](#)
- initializing
- CAN, [27-10](#)
- input clock, *See* CLKIN
- instruction bit scan ordering, [B-5](#)
- instruction register, [B-3](#), [B-5](#)
- instructions, [20-35](#)
- See also* instructions by name
- Integrated Drive Electronics interface, [24-1](#)
- INT_ENA (interrupt enable) bit, [32-135](#)
- interleaving
- of data in SPORT FIFO, [30-62](#)
 - SPORT data, [30-8](#)
- internal boot ROM, [20-35](#)
- internal clock generation (ICLKGEN) bit, [26-82](#)
- internal/external frame syncs, *See* frame sync
- internal frame sync generation (IFSGEN) bit, [26-82](#)
- internal memory, [20-7](#)
- interrupt channels, UART, [31-43](#)
- interrupt conditions, UART, [31-46](#)
- interrupt enable (INT_ENA) bit, [32-135](#)
- interrupt output, SPI, [28-25](#)
- interrupts
- CAN, [27-2](#)
 - SPI, [28-26](#), [28-52](#)
 - SPORT error, [30-40](#)
 - SPORT RX, [30-40](#), [30-65](#)
 - SPORT TX, [30-40](#), [30-62](#)
 - UART, [31-17](#)
- I/O interface to peripheral serial device, [30-3](#)
- I/O memory space, [20-10](#)
- IORDY_EN (IORDY enable) bit, [24-50](#)
- IRCLK bit, [30-57](#), [30-59](#)
- IrDA
- receiver, [31-15](#)
 - transmitter, [31-14](#)
- IrDA mode, [31-50](#)
- IREN bit, [31-50](#)
- IRFS bit, [30-34](#), [30-57](#), [30-59](#)
- IRPOL bit, [31-16](#)
- isochronous transfer enable (ISO_R) bit, [32-109](#)
- isochronous transfer enable (ISO_T) bit, [32-102](#)

Index

isochronous update enable
(ISO_UPDATE) bit, [32-82](#)
ISO_R (isochronous transfer enable) bit,
[32-109](#)
ISO_T (isochronous transfer enable) bit,
[32-102](#)
ISO_UPDATE (isochronous update
enable) bit, [32-82](#)
ITCLK bit, [30-51](#), [30-53](#)
ITFS bit, [30-21](#), [30-34](#), [30-51](#), [30-54](#)
ITU interlaced/progressive (ITU_TYPE)
bit, [26-82](#)
ITU output with internal blanking
(BLANKGEN) bit, [26-82](#)
ITU_TYPE (ITU interlaced/progressive)
bit, [26-82](#)

J

JTAG, [20-38](#), [B-1](#), [B-3](#), [B-5](#)

K

keypad
enable/disable, [22-4](#)
input matrix programmability, [22-4](#)
interface, [22-3](#)
interface overview, [22-1](#)
KPAD_CTL register, [22-4](#)
KPAD_PRESCALE register, [22-9](#)
operation, [22-2](#)
programming examples, [22-22](#)
programming model, [22-9](#)
registers
interrupt generation when X-Key
pressed, [22-18](#)
KPAD_CTL, [22-10](#), [A-13](#)
KPAD_MSEL, [22-10](#), [A-13](#)
KPAD_PRESCALE, [22-10](#), [A-13](#)
KPAD_ROWCOL, [22-10](#), [A-13](#)
KPAD_SOFTEVAL, [22-10](#), [A-13](#)

KPAD_STAT, [22-10](#), [A-13](#)
state diagram, [22-8](#)
keypad control (KPAD_CTL) register,
[22-10](#), [A-13](#)
keypad enable (KPAD_EN) bit, [22-10](#)
keypad interrupt status (KPAD_IRQ) bit,
[22-19](#)
keypad multiplier select (KPAD_MSEL)
register, [22-10](#), [22-16](#), [A-13](#)
keypad row-column (KPAD_ROWCOL)
register, [22-10](#), [22-16](#), [A-13](#)
keypad software evaluate
(KPAD_SOFTEVAL) register, [22-21](#)
key prescale (KPAD_PRESCALE) register,
[22-10](#), [22-14](#), [A-13](#)
key prescale value
(KPAD_PRESCALE_VAL) bits,
[22-14](#)
key press current status
(KPAD_PRESSED) bit, [22-19](#)
key software evaluate
(KPAD_SOFTEVAL) register, [22-10](#),
[A-13](#)
KPAD_COL (columns value pressed) bits,
[22-16](#)
KPAD_COLEN (column enable width)
bits, [22-10](#)
KPAD_CTL (keypad control) register,
[22-10](#), [A-13](#)
KPAD_EN (keypad enable) bit, [22-10](#)
KPAD_IRQ (keypad interrupt status) bit,
[22-19](#)
KPAD_IRQMODE (multikey press
interrupt enable) bits, [22-10](#)
KPAD_MROWCOL (multiple
row/column keypress) bits, [22-19](#)
KPAD_MSEL (keypad multiplier select)
register, [22-10](#), [22-16](#), [A-13](#)
KPAD_PRESCALE (key prescale) register,
[22-10](#), [22-14](#), [A-13](#)

KPAD_PRESCALE_VAL (key prescale value) bits, [22-14](#)
 KPAD_PRESSED (key press current status) bit, [22-19](#)
 KPAD_ROWCOL (keypad row-column) register, [22-10](#), [22-16](#), [A-13](#)
 KPAD_ROWEN (row enable width) bits, [22-10](#)
 KPAD_ROW (rows value pressed) bits, [22-16](#)
 KPAD_SOFTEVAL_E (software programmable force evaluate) bit, [22-21](#)
 KPAD_SOFTEVAL (keypad software evaluate) register, [22-21](#)
 KPAD_SOFTEVAL (key software evaluate) register, [22-10](#), [A-13](#)
 KPAD_STAT (keypad status) register, [22-10](#), [22-19](#), [A-13](#)

L

L1 data memory, [20-7](#)
 L1 instruction memory, [20-7](#)
 L1 scratchpad RAM, [20-7](#)
 LARFS bit, [30-37](#), [30-57](#), [30-60](#)
 large packet split (INCOMPTX_R) bit, [32-109](#)
 large packet split (INCOMPTX_RH) bit, [32-109](#)
 large packet split (INCOMPTX_T) bit, [32-102](#)
 late frame sync, [30-19](#), [30-37](#)
 LATFS bit, [30-37](#), [30-51](#), [30-55](#)
 line terminations, SPORT, [30-10](#)
 Line Track Overflow Error (LTERR_OVR) bit, [24-50](#), [24-56](#), [24-57](#)
 line track overflow (LTERR_OVR) bit, [26-89](#)

Line Track Underflow Error (LTERR_UNDR) bit, [24-50](#), [24-56](#), [24-57](#)
 line track underflow (LTERR_UNDR) bit, [26-89](#)
 little endian byte order, [29-52](#)
 load error indicator (DATAERROR_R) bit, [32-109](#)
 Lock Mechanism 0, [21-18](#)
 Lock Mechanism 1, [21-18](#)
 Lock Mechanism Select (LMECH) bit, [21-18](#)
 Logical Channel for Physical Channel x (LCHANPCx) field, [21-59](#)
 long response (CMD_L_RSP) bit, [23-24](#)
 LOOPBACK (loopback mode enable) bit, [31-33](#)
 loopback mode, UART, [31-33](#)
 LOSTARB bit, [29-15](#), [29-45](#)
 LOW_EVEN (lower limit for even bytes (luma) bits, [26-101](#)
 LOW_ODD (lower limit for odd bytes (chroma) bits, [26-101](#)
 low-speed device indicator (LSDEV) bit, [32-122](#), [32-124](#)
 low-speed EOF 1 (LS_EOF1) bits, [32-129](#)
 LRFS bit, [30-33](#), [30-35](#), [30-57](#), [30-60](#)
 LSBF bit, [28-45](#)
 LSB first (LSBF) bit, [28-45](#)
 LSBF (LSB first) bit, [28-45](#)
 LSDEV (low-speed device indicator) bit, [32-122](#), [32-124](#)
 LS_EOF1 (low-speed EOF 1) bits, [32-129](#)
 LT_ERR_OVR bit, [22-11](#), [22-20](#), [22-21](#)
 LTERR_OVR (line track overflow) bit, [26-89](#)
 LTERR_OVR (Line Track Overflow Error) bit, [24-50](#), [24-56](#), [24-57](#)
 LTERR_UNDR (line track underflow) bit, [26-89](#)

Index

LTERR_UNDR (Line Track Underflow Error) bit, [24-50](#), [24-56](#), [24-57](#)
LTFS bit, [30-21](#), [30-33](#), [30-35](#), [30-51](#), [30-55](#)
luma FIFO error (YFIFO_ERR) bit, [26-89](#)
Luma FIFO Overflow Error (YFIFO_ERR) bit, [24-50](#), [24-56](#), [24-57](#)

M

MAA bit, [27-48](#)

MAC

pins, [22-6](#)

MADDR[6:0] field, [29-44](#)

mailboxes, CAN, [27-5](#)

mailbox interrupts, CAN, [27-25](#)

manual

conventions, [-xlviii](#)

MASK_BUSYIRQ (mask not busy IRQ) bit, [25-23](#)

mask card detect (SCD_MSK) bit, [23-36](#)

mask not busy IRQ (MASK_BUSYIRQ) bit, [25-23](#)

MASK_RDRDY (mask read data ready) bit, [25-23](#)

mask read data ready (MASK_RDRDY) bit, [25-23](#)

MASK_WBEDGE (mask write buffer edge detect) bit, [25-23](#)

MASK_WBOVF (mask write buffer overflow) bit, [25-23](#)

MASK_WRDONE (mask write done) bit, [25-23](#)

mask write buffer edge detect (MASK_WBEDGE) bit, [25-23](#)

mask write buffer overflow (MASK_WBOVF) bit, [25-23](#)

mask write done (MASK_WRDONE) bit, [25-23](#)

Master mode initialization, [21-118](#)

master (MSTR) bit, [28-45](#)

Maximum Delay Register Updated interrupt enable, [21-45](#)

Maximum Delay Register Updated (MDRU) interrupt event, [21-32](#)

maximum individual packet size (MaxPktSize), [32-24](#), [32-25](#), [32-26](#), [32-27](#), [32-28](#), [32-29](#)

Maximum Position Register Updated interrupt enable, [21-45](#)

Maximum Position Register Updated (MPRU) interrupt event, [21-32](#)

MAX_PACKET_SIZE_R (USB max Rx data in frame) bits, [32-107](#)

MAX_PACKET_SIZE_T (USB max Tx data in frame) bits, [32-97](#)

MaxPktSize (maximum individual packet size), [32-24](#), [32-25](#), [32-26](#), [32-27](#), [32-28](#), [32-29](#)

MBCLK, [21-4](#)

MBIMn bit, [27-84](#)

MBPTR[4:0] field, [27-47](#)

MBRIFn bit, [27-86](#)

MBRIRQ bit, [27-49](#)

MBTIFn bit, [27-85](#)

MBTIRQ bit, [27-49](#)

MCCRM[1:0] field, [30-71](#)

MCDRXPE bit, [30-71](#)

MCCTXPE bit, [30-71](#)

MCMEN bit, [30-19](#), [30-71](#)

MCOMP bit, [29-20](#), [29-51](#)

MCOMP bit, [29-48](#), [29-50](#)

MCx bit, [27-73](#)

MDIR bit, [29-41](#), [29-43](#)

MDMA_TFRCNT (MDMA transfer count) bits, [24-63](#)

- MDMA_XFER_ON (multi-word DMA transfer in progress) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- MDn bit, [27-74](#)
- MDRUEN, [21-45](#)
- Meaning of CM Allocate Status, [21-143](#)
- Meaning of Transmission Status, [21-134](#)
- Media Oriented System Transport, [-xliii](#), [21-1](#)
- Media Transceiver module, [-xliii](#), [21-1](#)
- memory
 - architecture, [20-6](#)
 - configurations, [20-6](#)
 - external, [20-8](#)
 - EBIU, [20-8](#)
 - internal, [20-7](#)
 - I/O, [20-8](#)
 - I/O space, [20-10](#)
 - L1 data, [20-7](#)
 - L1 instruction, [20-7](#)
 - L1 scratchpad, [20-7](#)
 - moving data between SPORT and, [30-40](#)
 - on-chip, [20-7](#)
 - one-time-programmable, [20-10](#)
 - structure, [20-6](#)
- memory-mapped registers, *See* MMRs
- MEN bit, [29-41](#), [29-43](#)
- MERR bit, [29-19](#), [29-51](#)
- MERRM bit, [29-48](#), [29-50](#)
- MFD[3:0] field, [30-23](#), [30-71](#)
- MFLOW field, [21-70](#)
- MFS, [21-4](#)
- MH2LEN, [21-45](#)
- MII
 - pins, [22-6](#)
- MISO pin, [28-5](#), [28-6](#), [28-17](#), [28-19](#), [28-20](#), [28-22](#), [28-29](#)
- ML2HEN, [21-45](#)
- μ -law companding, [30-26](#), [30-31](#)
- MLF, [21-4](#)
- MLF analog pin, [21-3](#)
- MMCLK, [21-4](#)
- MMR
 - offset, [21-6](#)
- MMR Offset, [A-7](#)
- MMRs, [20-10](#)
 - address range, [A-3](#)
 - width, [A-3](#)
- MODE (DMA mode 0/1 selection) bit, [32-135](#)
- mode fault error, [28-24](#), [28-26](#)
- mode fault error (MODF) bit, [28-48](#)
- modes
 - broadcast, [28-12](#), [28-19](#), [28-20](#)
 - multichannel, [30-17](#)
 - serial port, [30-12](#)
 - SPI master, [28-20](#), [28-26](#)
 - SPI slave, [28-20](#), [28-29](#)
 - UART DMA, [31-25](#)
 - UART non-DMA, [31-23](#)
- MODF bit, [28-24](#), [28-48](#)
- MODF (mode fault error) bit, [28-48](#)
- MOSI pin, [28-5](#), [28-6](#), [28-17](#), [28-19](#), [28-20](#), [28-22](#), [28-30](#)
- MOST®, [-xliii](#), [21-1](#)
- MOST® NetInterface, [21-1](#)
- moving data, serial port, [30-40](#)
- moving window enable (MWE) bit, [23-37](#)
- MPIVDD, [21-5](#)
- M (PLL multiplier select) bits, [32-132](#)
- MPROG bit, [29-15](#), [29-45](#)
- MPRUEN, [21-45](#)
- MRB bit, [27-48](#)
- MRTS (manual request to send) bit, [31-33](#)
- MRX, [21-3](#), [21-4](#)
- MRX input pin, [21-20](#)
- MRXONB, [21-4](#)

Index

- MRXONB High to Low interrupt enable, [21-45](#)
- MRXONB High to Low (MH2L) interrupt event, [21-34](#)
- MRXONB Low to High interrupt enable, [21-45](#)
- MRXONB Low to High (ML2H) interrupt event, [21-34](#)
- MSTR bit, [28-22](#), [28-45](#)
- MSTR (master) bit, [28-45](#)
- MTX, [21-4](#)
- MTXON, [21-4](#)
- multichannel frame, [30-22](#)
- multichannel frame delay field, [30-23](#)
- multichannel mode, [30-17](#)
 - enable/disable, [30-19](#)
 - frame syncs, [30-20](#)
 - SPORT, [30-20](#)
- multichannel operation, SPORT, [30-17](#) to [30-26](#)
- multi-DMA transfer done interrupt mask (MULTI_DONE_MASK) bit, [24-56](#)
- multi-DMA transfer done interrupt status (MULTI_DONE_INT) bit, [24-58](#)
- MULTI_DONE_INT (multi-DMA transfer done interrupt status) bit, [24-58](#)
- MULTI_DONE_MASK (multi-DMA transfer done interrupt mask) bit, [24-56](#)
- multikey press interrupt enable (KPAD_IRQMODE) bits, [22-10](#)
- multiple row/column keypress (KPAD_MROWCOL) bits, [22-19](#)
- multiple slave SPI systems, [28-12](#)
- multiplexed with GPIO, [21-3](#)
- MULTI_START (start multi-DMA Op) bit, [24-50](#)
- MULTI_TERM_INT (device terminate multi-DMA transfer interrupt status) bit, [24-58](#)
- MULTI_TERM_MASK (device terminate multi-DMA transfer interrupt mask) bit, [24-56](#)
- multi-word DMA transfer in progress (MDMA_XFER_ON) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- multi-word DMA transfer in progress (MULTI_XFER_ON) bit, [24-52](#)
- MULTI_XFER_ON (multi-word DMA transfer in progress) bit, [24-52](#)
- MVIP-90, [30-27](#)
- MWE (moving window enable) bit, [23-37](#)
- MXEGND, [21-5](#)
- MXI, [21-4](#)
- MXO, [21-4](#)
- MXVR, [-xliii](#), [21-1](#)
- MXVR_AADDR, [21-56](#)
- MXVR_AADDR (MXVR alternate address) register, [21-56](#)
- MXVR allocation table (MXVR_ALLOC_x) registers, [21-57](#)
- MXVR Allocation Table Registers, [21-56](#)
- MXVR_ALLOC_x, [21-56](#)
- MXVR_ALLOC_x (MXVR allocation table) registers, [21-57](#)
- MXVR alternate address (MXVR_AADDR) register, [21-56](#)
- MXVR Alternate Address Register, [21-56](#)
- MXVR_AP_CTL, [21-77](#)
- MXVR_AP_CTL (MXVR asynchronous packet control) register, [21-77](#)
- MXVR_AP_CTL register, [21-77](#)
- MXVR_APRB_CURR_ADDR (MXVR asynchronous packet receive buffer current address) register, [21-81](#)

- MXVR_APRB_CURR_ADDR register, [21-81](#)
- MXVR_APRB_START_ADDR (MXVR asynchronous packet receive buffer start address) register, [21-80](#)
- MXVR_APRB_START_ADDR register, [21-80](#)
- MXVR_APTB_CURR_ADDR (MXVR asynchronous packet transmit buffer current address) register, [21-83](#)
- MXVR_APTB_CURR_ADDR register, [21-83](#)
- MXVR_APTB_START_ADDR (MXVR asynchronous packet transmit buffer start address) register, [21-82](#)
- MXVR_APTB_START_ADDR registers, [21-82](#)
- MXVR asynchronous packet control (MXVR_AP_CTL) register, [21-77](#)
- MXVR Asynchronous Packet Control Register, [21-77](#)
- MXVR asynchronous packet receive buffer current address (MXVR_APRB_CURR_ADDR) register, [21-81](#)
- MXVR Asynchronous Packet Receive Buffer Current Address Register, [21-81](#)
- MXVR asynchronous packet receive buffer start address (MXVR_APRB_START_ADDR) register, [21-80](#)
- MXVR Asynchronous Packet Receive Buffer Start Address Register, [21-80](#)
- MXVR asynchronous packet transmit buffer current address (MXVR_APTB_CURR_ADDR) register, [21-83](#)
- MXVR Asynchronous Packet Transmit Buffer Current Address Register, [21-83](#)
- MXVR asynchronous packet transmit buffer start address (MXVR_APTB_START_ADDR) register, [21-82](#)
- MXVR Asynchronous Packet Transmit Buffer Start Address Register, [21-82](#)
- MXVR Bit Clock, [21-4](#)
- MXVR_BLOCK_CNT, [21-98](#)
- MXVR_BLOCK_CNT (MXVR block counter) register, [21-99](#)
- MXVR block counter (MXVR_BLOCK_CNT) register, [21-99](#)
- MXVR Block Counter Register, [21-98](#)
- MXVR_CM_CTL (MXVR control message control) register, [21-84](#)
- MXVR_CM_CTL register, [21-83](#)
- MXVR_CMRB_CURR_ADDR, [21-87](#)
- MXVR_CMRB_CURR_ADDR (MXVR control message receive buffer current address) register, [21-87](#)
- MXVR_CMRB_START_ADDR, [21-86](#)
- MXVR_CMRB_START_ADDR (MXVR control message receive buffer start address) register, [21-86](#)
- MXVR_CMTB_CURR_ADDR, [21-89](#)
- MXVR_CMTB_CURR_ADDR (MXVR control message transmit buffer current address) register, [21-89](#)
- MXVR_CMTB_START_ADDR, [21-88](#)
- MXVR_CMTB_START_ADDR (MXVR control message transmit buffer start address) register, [21-88](#)
- MXVR_CONFIG, [21-12](#)
- MXVR_CONFIG (MXVR configuration) register, [21-12](#)

Index

- MXVR configuration (MXVR_CONFIG) register, [21-12](#)
- MXVR Configuration Register, [21-12](#)
- MXVR control message control (MXVR_CM_CTL) register, [21-84](#)
- MXVR Control Message Control Register, [21-83](#)
- MXVR control message receive buffer current address (MXVR_CMRB_CURR_ADDR) register, [21-87](#)
- MXVR Control Message Receive Buffer Current Address Register, [21-87](#)
- MXVR control message receive buffer start address (MXVR_CMRB_START_ADDR) register, [21-86](#)
- MXVR Control Message Receive Buffer Start Address Register, [21-86](#)
- MXVR Control Message Transmit Buffer (CMTB), [21-131](#)
- MXVR control message transmit buffer current address (MXVR_CMTB_CURR_ADDR) register, [21-89](#)
- MXVR Control Message Transmit Buffer Current Address Register, [21-89](#)
- MXVR control message transmit buffer start address (MXVR_CMTB_START_ADDR) register, [21-88](#)
- MXVR Control Message Transmit Buffer Start Address Register, [21-88](#)
- MXVR Crystal Input, [21-4](#)
- MXVR Crystal Output, [21-4](#)
- MXVR_DELAY, [21-51](#)
- MXVR_DELAY (MXVR node frame delay) register, [21-51](#)
- MXVR DMA Channel x Current Address Registers, [21-72](#)
- MXVR DMA Channel x Current Transfer Count Registers, [21-76](#)
- MXVR DMA Channel x Start Address Registers, [21-70](#)
- MXVR DMA Channel x Transfer Count Registers, [21-73](#)
- MXVR_DMAx_CONFIG (MXVR DMAx data configuration) registers, [21-69](#)
- MXVR DMAx Configuration Registers, [21-60](#)
- MXVR_DMAx_COUNT, [21-73](#)
- MXVR_DMAx_CURR_ADDR, [21-72](#), [21-73](#)
- MXVR_DMAx_CURR_ADDR (MXVR sync data DMAx current address) registers, [21-74](#)
- MXVR_DMAx_CURR_COUNT, [21-76](#)
- MXVR_DMAx_CURR_COUNT (MXVR sync data DMAx current loop count) registers, [21-76](#)
- MXVR_DMAx_START_ADDR, [21-70](#), [21-73](#)
- MXVR_DMAx_START_ADDR (MXVR sync data DMAx start address) registers, [21-70](#), [21-73](#)
- MXVR Enable (MXVREN) bit, [21-12](#)
- MXVR_FRAME_CNT_x (MXVR frame counter) registers, [21-95](#)
- MXVR_FRAME_CNT_x register, [21-94](#)
- MXVR frame counter (MXVR_FRAME_CNT_x) registers, [21-95](#)
- MXVR Frame Counter Registers, [21-94](#)
- MXVR Frame Sync, [21-4](#)
- MXVR_GADDR, [21-55](#)
- MXVR_GADDR (MXVR group address) register, [21-55](#)
- MXVR group address (MXVR_GADDR) register, [21-55](#)

- MXVR Group Address Register, [21-55](#)
- MXVR_INT_EN_0, [21-43](#)
- MXVR_INT_EN_0 (MXVR interrupt enable) register 0, [21-44](#)
- MXVR_INT_EN_1, [21-46](#)
- MXVR_INT_EN_1 (MXVR interrupt enable) register 1, [21-47](#)
- MXVR interrupt enable (MXVR_INT_EN_0) register 0, [21-44](#)
- MXVR interrupt enable (MXVR_INT_EN_1) register 1, [21-47](#)
- MXVR Interrupt Enable Register 0, [21-43](#)
- MXVR Interrupt Enable Register 1, [21-46](#)
- MXVR interrupt status (MXVR_INT_STAT_0) register 0, [21-30](#)
- MXVR interrupt status (MXVR_INT_STAT_1) register 1, [21-40](#)
- MXVR Interrupt Status Register 0, [21-29](#)
- MXVR Interrupt Status Register_1, [21-40](#)
- MXVR_INT_STAT_0, [21-29](#)
- MXVR_INT_STAT_0 (MXVR interrupt status) register 0, [21-30](#)
- MXVR_INT_STAT_1, [21-40](#)
- MXVR_INT_STAT_1 (MXVR interrupt status) register 1, [21-40](#)
- MXVR_LADDR, [21-54](#)
- MXVR_LADDR (MXVR logical address) register, [21-54](#)
- MXVR logical address (MXVR_LADDR) register, [21-54](#)
- MXVR Logical Address Register, [21-54](#)
- MXVR Master Clock, [21-4](#)
- MXVR Master Mode/Slave Mode Select (MMSM) bit, [21-13](#)
- MXVR_MAX_DELAY, [21-53](#)
- MXVR_MAX_DELAY (MXVR maximum node frame delay) register, [21-53](#)
- MXVR maximum node frame delay (MXVR_MAX_DELAY) register, [21-53](#)
- MXVR Maximum Node Frame Delay Register, [21-53](#)
- MXVR maximum node position (MXVR_MAX_POSITION) register, [21-50](#)
- MXVR Maximum Node Position Register, [21-50](#)
- MXVR_MAX_POSITION, [21-50](#)
- MXVR_MAX_POSITION (MXVR maximum node position) register, [21-50](#)
- MXVR Memory Map, [21-5](#), [A-7](#)
- MXVR MMR address offsets, [21-5](#)
- MXVR node frame delay (MXVR_DELAY) register, [21-51](#)
- MXVR Node Frame Delay Register, [21-51](#)
- MXVR node position (MXVR_POSITION) register, [21-49](#)
- MXVR Node Position Register, [21-48](#)
- MXVR Parity select (PARITY) bit, [21-16](#)
- MXVR_PAT_DATA_x (MXVR pattern data) registers, [21-92](#)
- MXVR_PAT_DATA_x registers, [21-92](#)
- MXVR_PAT_EN_x (MXVR pattern enable) registers, [21-93](#)
- MXVR_PAT_EN_x registers, [21-93](#)
- MXVR pattern data (MXVR_PAT_DATA_x) registers, [21-92](#)
- MXVR Pattern Data Registers, [21-92](#)
- MXVR pattern enable (MXVR_PAT_EN_x) registers, [21-93](#)
- MXVR Pattern Enable Register, [21-93](#)

Index

- MXVR Pattern Matching, [21-91](#)
- MXVR PHY Receiver Receiving Light, [21-4](#)
- MXVR_PLL_CTL_0 register, Initialization, [21-119](#)
- MXVR_POSITION, [21-48](#)
- MXVR_POSITION (MXVR node position) register, [21-49](#)
- MXVR Power Supply Pins, [21-5](#)
- MXVR Power-Up PHY Transmitter, [21-4](#)
- MXVR Power-Up PHY Transmitter (MTXON) bit, [21-16](#)
- MXVR Receive Data, [21-4](#)
- MXVR Register
 - Allocation Table x, [A-7](#)
 - Alternate Address, [A-7](#)
 - Control Message Control, [A-11](#)
 - Control Message RX Buffer Current Address, [A-11](#)
 - Control Message RX Buffer Start Address, [A-11](#)
 - Control Message TX Buffer Current Address, [A-11](#)
 - Control Message TX Buffer Start Address, [A-11](#)
 - Group Address, [A-7](#)
 - Logical Address, [A-7](#)
 - Maximum Node Frame Delay, [A-7](#)
 - Maximum Node Position, [A-7](#)
 - Node Frame Delay, [A-7](#)
 - Node Position, [A-7](#)
 - Pattern Data 0, [A-11](#)
 - Pattern Data 1, [A-11](#)
 - Pattern Enable 0, [A-11](#)
 - Pattern Enable 1, [A-11](#)
 - Phase Lock Loop Control, [A-12](#)
 - Remote Read Buffer Current Address, [A-11](#)
 - Remote Read Buffer Start Address, [A-11](#)
 - Routing x, [A-12](#)
 - State 0, [A-7](#)
 - State 1, [A-7](#)
- Synchronous Data DMA0
 - Configuration, [A-8](#)
- Synchronous Data DMA0 Current Address, [A-8](#)
- Synchronous Data DMA0 Current Loop Count, [A-8](#)
- Synchronous Data DMA0 Loop Count, [A-8](#)
- Synchronous Data DMA0 Start Address, [A-8](#)
- Synchronous Data DMA1
 - Configuration, [A-9](#)
- Synchronous Data DMA1 Current Address, [A-9](#)
- Synchronous Data DMA1 Current Loop Count, [A-9](#)
- Synchronous Data DMA1 Loop Count, [A-9](#)
- Synchronous Data DMA1 Start Address, [A-9](#)
- Synchronous Data DMA2
 - Configuration, [A-9](#)
- Synchronous Data DMA2 Current Address, [A-9](#)
- Synchronous Data DMA2 Current Loop Count, [A-9](#)
- Synchronous Data DMA2 Loop Count, [A-9](#)
- Synchronous Data DMA2 Start Address, [A-9](#)
- Synchronous Data DMA3
 - Configuration, [A-9](#)
- Synchronous Data DMA3 Current Address, [A-9](#)
- Synchronous Data DMA3 Current Loop Count, [A-9](#)
- Synchronous Data DMA3 Loop Count, [A-9](#)

- Synchronous Data DMA3 Start Address, [A-9](#)
- Synchronous Data DMA4
 - Configuration, [A-9](#)
- Synchronous Data DMA4 Current Address, [A-10](#)
- Synchronous Data DMA4 Current Loop Count, [A-10](#)
- Synchronous Data DMA4 Loop Count, [A-10](#)
- Synchronous Data DMA4 Start Address, [A-10](#)
- Synchronous Data DMA5
 - Configuration, [A-10](#)
- Synchronous Data DMA5 Current Address, [A-10](#)
- Synchronous Data DMA5 Current Loop Count, [A-10](#)
- Synchronous Data DMA5 Loop Count, [A-10](#)
- Synchronous Data DMA5 Start Address, [A-10](#)
- Synchronous Data DMA6
 - Configuration, [A-10](#)
- Synchronous Data DMA6 Current Address, [A-10](#)
- Synchronous Data DMA6 Current Loop Count, [A-10](#)
- Synchronous Data DMA6 Loop Count, [A-10](#)
- Synchronous Data DMA6 Start Address, [A-10](#)
- Synchronous Data DMA7
 - Configuration, [A-10](#)
- Synchronous Data DMA7 Current Loop Count, [A-11](#)
- Synchronous Data DMA7 Loop Count, [A-11](#)
- Synchronous Data DMA7 Start Address, [A-10](#)
- Synchronous Data DMAx Current Address, [A-11](#)
- Synchronous Data Logical Channel Assignment x, [A-8](#)
- MXVR registers
 - list of, [21-6](#)
- MXVR remote read buffer current address (MXVR_RRDB_CURR_ADDR)
 - register, [21-91](#)
- MXVR Remote Read Buffer Current Address Register, [21-91](#)
- MXVR remote read buffer start address (MXVR_RRDB_START_ADDR)
 - register, [21-90](#)
- MXVR Remote Read Buffer Start Address Register, [21-90](#)
- MXVR_ROUTING_0 (MXVR routing 0)
 - register, [21-96](#)
- MXVR routing 0 (MXVR_ROUTING_0)
 - register, [21-96](#)
- MXVR Routing Registers, [21-95](#)
- MXVR_ROUTING_x register,
 - Initialization, [21-119](#)
- MXVR_ROUTING_x registers, [21-95](#)
- MXVR_RRDB_CURR_ADDR, [21-91](#)
- MXVR_RRDB_CURR_ADDR (MXVR remote read buffer current address)
 - register, [21-91](#)
- MXVR_RRDB_START_ADDR, [21-90](#)
- MXVR_RRDB_START_ADDR (MXVR remote read buffer start address)
 - register, [21-90](#)
- MXVR Signal Pins, [21-4](#)
- MXVR signal pins, [21-3](#)
- MXVR_STATE_0, [21-19](#)
- MXVR_STATE_1, [21-19](#)
- MXVR State Registers, [21-19](#)
- MXVR_STATE_x state registers, [21-19](#)

Index

- MXVR sync data DMAx current address (MXVR_DMAx_CURR_ADDR) registers, [21-74](#)
- MXVR sync data DMAx current loop count (MXVR_DMAx_CURR_COUNT) registers, [21-76](#)
- MXVR sync data DMAx start address (MXVR_DMAx_START_ADDR) registers, [21-70](#), [21-73](#)
- MXVR synchronous logical channel assignment (MXVR_SYNC_LCHAN_x) registers, [21-59](#)
- MXVR Synchronous Logical Channel Assignment Registers, [21-58](#)
- MXVR_SYNC_LCHAN_x, [21-58](#)
- MXVR_SYNC_LCHAN_x (MXVR synchronous logical channel assignment) registers, [21-59](#)
- MXVR Transmit Data, [21-4](#)
- MXVR Transmit Data Pin Enable (MTXEN) bit, [21-16](#)

- N**
- NA2IEN, [21-44](#)
- NAK bit, [29-37](#), [29-38](#)
- NAK_TIMEOUT_H (EP halted after a NAK) bit, [32-98](#), [32-102](#)
- NAND address (NFC_ADDR) register, [25-19](#), [25-27](#), [A-19](#)
- NAND command (NFC_CMD) register, [25-19](#), [25-28](#), [A-19](#)
- NAND control (NFC_CTL) register, [25-18](#), [25-20](#), [A-18](#)
- NAND data read (NFC_DATA_RD) register, [25-19](#), [25-29](#), [A-19](#)
- NAND data width (NWIDTH) bit, [25-20](#)
- NAND data write (NFC_DATA_WR) register, [25-19](#), [25-29](#), [A-19](#)
- NAND ECC count (NFC_COUNT) register, [25-18](#), [25-25](#), [A-18](#)
- NAND ECC reset (NFC_RST) register, [25-18](#), [25-25](#), [A-18](#)
- NAND flash controller, [20-9](#)
 - additional operations, [25-10](#)
 - NFC accesses, [25-6](#)
 - NFC error detection, [25-11](#)
 - NFC external interface, [25-4](#)
 - NFC interface block diagram, [25-4](#)
 - overview, [25-1](#)
 - page read, [25-9](#)
 - page write, [25-8](#)
- NAND interrupt mask (NFC_IRQMASK) register, [25-18](#), [25-23](#), [A-18](#)
- NAND interrupt status (NFC_IRQSTAT) register, [25-18](#), [25-22](#), [A-18](#)
- NAND page control (NFC_PGCTL) register, [25-18](#), [25-26](#), [A-18](#)
- NAND read data (NFC_READ) register, [25-18](#), [25-27](#), [A-18](#)
- NAND status (NFC_STAT) register, [25-18](#), [25-21](#), [A-18](#)
- NBUSYIRQ (not busy IRQ) bit, [25-22](#)
- NBUSY (not busy) bit, [25-21](#)
- NetServices Layer 1, [21-1](#)
- Network Active (NACT) bit, [21-20](#)
- Network Active to Inactive interrupt enable, [21-44](#)
- Network Active to Inactive (NA2I) interrupt event, [21-31](#)
- Network Activity Detection, [21-115](#)
- Network Activity (NACT) bit, [21-31](#)
- Network Inactive to Active interrupt enable, [21-44](#)
- Network Inactive to Active (NI2A) interrupt event, [21-31](#)
- Network Initialization, [21-121](#)
- Network Lock, [21-121](#)
- network slave, [21-13](#)

- network timing master, [21-13](#)
 - NFC
 - features, [25-2](#)
 - NFC_ADDR (NAND address) register, [25-19, 25-27, A-19](#)
 - NFC_CMD (NAND command) register, [25-19, 25-28, A-19](#)
 - NFC_COUNT (NAND ECC count) register, [25-18, 25-25, A-18](#)
 - NFC_CTL (NAND control) register, [25-18, 25-20, A-18](#)
 - NFC_DATA_RD (NAND data read) register, [25-19, 25-29, A-19](#)
 - NFC_DATA_WR (NAND data write) register, [25-19, 25-29, A-19](#)
 - NFC_ECC0 (NAND ECC) register 0, [25-18, 25-24, A-18](#)
 - NFC_ECC1 (NAND ECC) register 1, [25-18, 25-24, A-18](#)
 - NFC_ECC1 (NAND ECC) register 2, [25-24](#)
 - NFC_ECC2 (NAND ECC) register 2, [25-18, A-18](#)
 - NFC_ECC3 (NAND ECC) register 3, [25-18, 25-24, A-18](#)
 - NFC_ECCx (NAND ECC) registers, [25-18, 25-24, A-18](#)
 - NFC_IRQMASK (NAND interrupt mask) register, [25-18, 25-23, A-18](#)
 - NFC_IRQSTAT (NAND interrupt status) register, [25-18, 25-22, A-18](#)
 - NFC_PGCTL (NAND page control) register, [25-18, 25-26, A-18](#)
 - NFC_READ (NAND read data) register, [25-18, 25-27, A-18](#)
 - NFC_RST (NAND ECC reset) register, [25-18, 25-25, A-18](#)
 - NFC_STAT (NAND status) register, [25-18, 25-21, A-18](#)
 - NI2AEN, [21-44](#)
 - Node Initialization, [21-117](#)
 - nominal bit rate, CAN, [27-12](#)
 - nominal bit time, CAN, [27-11](#)
 - Normal Control Message Receive Enable (NCMRXEN) bit, [21-15](#)
 - Normal Control Message Transmission, [21-135](#)
 - Normal Control Message Transmit Buffer Entry Field Offsets, [21-136](#)
 - normal frame sync mode, [30-37](#)
 - normal timing, serial port, [30-37](#)
 - not busy IRQ (NBUSYIRQ) bit, [25-22](#)
 - no TxPktRdy for IN token (OVERRUN_R) bit, [32-109](#)
 - no TxPktRdy for IN token (UNDERRUN_T) bit, [32-102](#)
 - NWIDTH (NAND data width) bit, [25-20](#)
- ## O
- OE (overrun error) bit, [31-36, 31-37](#)
 - on-chip memory, [20-7](#)
 - one/two DMA channel modes (DMACFG) bit, [26-83](#)
 - open drain drivers, [28-1](#)
 - open drain outputs, [28-20](#)
 - open drain output (SD_CMD_OD) bit, [23-22](#)
 - operating modes
 - active, [20-33](#)
 - deep sleep, [20-34](#)
 - full on, [20-33](#)
 - hibernate state, [20-34](#)
 - sleep, [20-33](#)
 - operating mode (XFR_TYPE) bits, [26-82](#)
 - OPSSn bit, [27-77](#)
 - optical Phy, [21-1](#)
 - OVERRUN_R (no TxPktRdy for IN token) bit, [32-109](#)

Index

P

PAB

- errors generated by SPORT, [30-41](#)
- PACKEN (pack/unpack enable) bit, [26-83](#)
- packet transaction status
 - (STATUSPKT_H) bit, [32-98](#)
- packing, serial port, [30-26](#)
- pack/unpack enable (PACKEN) bit, [26-83](#)
- page read pending (PG_RD_STAT) bit, [25-21](#)
- page read start (PG_RD_START) bit, [25-26](#)
- page write done (WR_DONE) bit, [25-22](#)
- page write pending (PG_WR_STAT) bit, [25-21](#)
- page write start (PG_WR_START) bit, [25-26](#)
- parity calculation result0 (ECC0) bits, [25-24](#)
- parity calculation result1 (ECC1) bits, [25-24](#)
- parity calculation result2 (ECC2) bits, [25-24](#)
- parity calculation result3 (ECC3) bits, [25-24](#)
- Parity Error interrupt enable, [21-45](#)
- Parity Error (PERR) interrupt event, [21-33](#)
- Pattern 0 Registers (PR0), [21-91](#)
- Pattern 1 Registers (PR1), [21-91](#)
- PC133 SDRAM controller, [20-14](#)
- PD_SDDAT3 (pull-down SD_DATA3) bit, [23-37](#)
- PEN (parity enable) bit, [31-30](#)
- PE (parity error) bit, [31-36](#), [31-37](#)
- peripherals, [20-3](#)
 - and buses, [20-4](#)
 - DMA support, [20-4](#)
 - list of, [20-3](#)
- PERREN, [21-45](#)

- PFx pin, [28-10](#)
- PG_RD_START (page read start) bit, [25-26](#)
- PG_RD_STAT (page read pending) bit, [25-21](#)
- PG_SIZE (page size) bit, [25-20](#)
- PG_WR_START (page write start) bit, [25-26](#)
- PG_WR_STAT (page write pending) bit, [25-21](#)
- Phy Receiver, [21-116](#)
- Phy Transmitter, [21-16](#)
- pin interrupt mask set
 - (PINTx_MASK_SET) registers, [21-100](#), [21-107](#), [21-110](#)
- pin interrupt x (PIQx) bits, [21-100](#), [21-107](#), [21-110](#)
- pin terminations, SPORT, [30-10](#)
- PINTx_MASK_SET (pin interrupt mask set) registers, [21-100](#), [21-107](#), [21-110](#)
- PIO_CSTATE (PIO mode state machine current state) bits, [24-61](#)
- PIO-DMA enable (PIO_USE_DMA) bit, [24-50](#)
- PIO_DONE_INT (PIO transfer done interrupt status) bit, [24-58](#)
- PIO_DONE_MASK (PIO transfer done interrupt mask) bit, [24-56](#)
- PIO mode state machine current state (PIO_CSTATE) bits, [24-61](#)
- PIO_START (start PIO/Reg Op) bit, [24-50](#)
- PIO_TFRCNT (PIO transfer count) bits, [24-62](#)
- PIO transfer done interrupt mask (PIO_DONE_MASK) bit, [24-56](#)
- PIO transfer done interrupt status (PIO_DONE_INT) bit, [24-58](#)

- PIO transfer in progress (PIO_XFER_ON) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- PIO_USE_DMA (PIO-DMA enable) bit, [24-50](#)
- PIO_XFER_ON (PIO transfer in progress) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- PIQx (pin interrupt x) bits, [21-100](#), [21-107](#), [21-110](#)
- PJx pin, [28-10](#)
- PLLCLKOE (PLL clock output enable) bit, [32-132](#)
- PLL clock output enable (PLLCLKOE) bit, [32-132](#)
- PLL multiplier select (M) bits, [32-132](#)
- PLL stable indicator (PLL_STABLE) bit, [32-132](#)
- PLL_STABLE (PLL stable indicator) bit, [32-132](#)
- PLL Start-Up Sequence, [21-120](#)
- POLC bits, [26-82](#)
- POLS bits, [26-82](#)
- port connection, SPORT, [30-8](#)
- port F
 - and SPI, [28-4](#)
- port function enable (PORTx_FER) registers, [21-112](#), [21-114](#)
- port pins, [28-47](#)
- port pins, test access, [B-3](#)
- ports, [20-15](#)
- port x bit y (Pxy) bits, [21-112](#), [21-114](#)
- PORTx_FER (port function enable) registers, [21-112](#), [21-114](#)
- POSITION field, [21-49](#)
- Position Register Updated interrupt enable, [21-45](#)
- Position Register Updated (PRU) interrupt event, [21-31](#)
- power management, [20-32](#)
- power on (PWR_ON) bits, [23-22](#)
- power save enable (PWR_SV_E) bit, [23-23](#)
- PPI_STATUS (PPI status register), [24-50](#), [24-56](#), [24-57](#), [26-100](#), [26-101](#)
- PPI status register (PPI_STATUS), [24-50](#), [24-56](#), [24-57](#), [26-100](#), [26-101](#)
- preamble error detected (ERR_DET) bit, [26-89](#)
- preamble error not corrected (ERR_NCOR) bit, [26-89](#)
- prescale[6:0] field, [29-36](#)
- private instructions, [B-5](#)
- processor block diagram, [20-5](#)
- propagation segment, CAN, [27-12](#)
- PROTOCOL_R (Rx protocol type) bits, [32-119](#)
- PROTOCOL_T (Tx protocol type) bits, [32-117](#)
- PRUEN, [21-45](#)
- PSSE bit, [28-21](#), [28-45](#)
- PSSE (slave select enable) bit, [28-45](#)
- public instructions, [B-5](#)
- public JTAG scan instructions, [B-5](#)
- pull-down SD_DATA3 (PD_SDDAT3) bit, [23-37](#)
- pull-up SD_DATA3 (PUP_SDDAT3) bit, [23-37](#)
- pull-up SD_DATA (PUP_SDDAT) bit, [23-37](#)
- PUP_SDDAT3 (pull-up SD_DATA3) bit, [23-37](#)
- PUP_SDDAT (pull-up SD_DATA) bit, [23-37](#)
- PWR_ON (power on) bits, [23-22](#)
- PWR_SV_E (power save enable) bit, [23-23](#)

Index

Pxy (port x bit y) bits, [21-112](#), [21-114](#)

R

RBSY bit, [28-48](#)

RBSY flag, [28-25](#)

RBSY (receive error) bit, [28-48](#)

RCKFE bit, [30-35](#), [30-57](#), [30-60](#)

RCVDATA16[15:0] field, [29-55](#)

RCVDATA8[7:0] field, [29-54](#)

RCVFLUSH bit, [29-45](#), [29-46](#)

RCVINTLEN bit, [29-45](#), [29-46](#)

RCVSERV bit, [29-18](#), [29-19](#), [29-51](#)

RCVSERVM bit, [29-48](#), [29-49](#)

RCVSTAT[1:0] field, [29-17](#), [29-47](#)

RD_DLY (read strobe delay) bits, [25-20](#)

RD_RDY (read data ready) bit, [25-22](#)

RDTYPE[1:0] field, [30-30](#), [30-57](#), [30-59](#)

read data ready (RD_RDY) bit, [25-22](#)

read strobe delay (RD_DLY) bits, [25-20](#)

read wait request (RWR) bit, [23-38](#)

READY_PAUSE (select ready to pause) bits, [24-69](#)

real-time clock, *See* RTC

REC bit, [27-47](#)

receive active mask (RX_ACT_MASK) bit, [23-33](#)

receive active (RX_ACT) bit, [23-30](#)

receive buffer[7:0] field, [31-42](#)

receive data[15:0] field, [30-65](#)

receive data[31:16] field, [30-65](#)

receive data available mask

(RX_DAT_RDY_MASK) bit, [23-33](#)

receive data available (RX_DAT_RDY) bit, [23-31](#)

receive data buffer[15:0] field, [28-49](#)

received byte count in EP0 FIFO (EP0_RX_COUNT) bits, [32-115](#)

receive error (RBSY) bit, [28-48](#)

receive FIFO, SPORT, [30-63](#)

receive FIFO empty mask (RX_FIFO_ZERO_MASK) bit, [23-33](#)

receive FIFO empty (RX_FIFO_ZERO) bit, [23-31](#)

receive FIFO full mask (RX_FIFO_FULL_MASK) bit, [23-33](#)

receive FIFO full (RX_FIFO_FULL) bit, [23-31](#)

receive FIFO status mask (RX_FIFO_STAT_MASK) bit, [23-33](#)

receive FIFO status (RX_FIFO_STAT) bit, [23-30](#)

receive message lost interrupt, CAN, [27-27](#)

receive overrun (RX_OVERRUN) bit, [23-30](#)

Receive Synchronous Boundary (RSB) field, [21-26](#)

receive underrun mask (RX_UNDERRUN_MASK) bit, [23-33](#)

receive underrun status (RX_UNDERRUN_STAT) bit, [23-32](#)

Receiving Control Message (CMRX) bit, [21-24](#)

reception error, SPI, [28-25](#)

Registers

[MXVR_AADDR](#), [A-7](#)
[MXVR_ALLOC_0](#), [A-7](#)
[MXVR_ALLOC_1](#), [A-7](#)
[MXVR_ALLOC_10](#), [A-8](#)
[MXVR_ALLOC_11](#), [A-8](#)
[MXVR_ALLOC_12](#), [A-8](#)
[MXVR_ALLOC_13](#), [A-8](#)
[MXVR_ALLOC_14](#), [A-8](#)
[MXVR_ALLOC_2](#), [A-7](#)
[MXVR_ALLOC_3](#), [A-7](#)
[MXVR_ALLOC_4](#), [A-7](#)
[MXVR_ALLOC_5](#), [A-7](#)
[MXVR_ALLOC_7](#), [A-7](#)
[MXVR_ALLOC_8](#), [A-7](#)
[MXVR_ALLOC_9](#), [A-8](#)
[MXVR_AP_CTL](#), [A-11](#)
[MXVR_APRB_CURR_ADDR](#), [A-11](#)
[MXVR_APRB_START_ADDR](#), [A-11](#)
[MXVR_APTB_CURR_ADDR](#), [A-11](#)
[MXVR_APTB_START_ADDR](#), [A-11](#)
[MXVR_CM_CTL](#), [A-11](#)
[MXVR_CMRB_CURR_ADDR](#), [A-11](#)
[MXVR_CMRB_START_ADDR](#), [A-11](#)
[MXVR_CMTB_CURR_ADDR](#), [A-11](#)
[MXVR_CMTB_START_ADDR](#), [A-11](#)
[MXVR_CONFIG](#), [A-7](#)
[MXVR_DELAY](#), [A-7](#)
[MXVR_DMA0_CONFIG](#), [A-8](#)
[MXVR_DMA0_COUNT](#), [A-8](#)
[MXVR_DMA0_CURR_ADDR](#), [A-8](#)
[MXVR_DMA0_CURR_COUNT](#), [A-8](#)
[MXVR_DMA0_START_ADDR](#), [A-8](#)
[MXVR_DMA1_CONFIG](#), [A-9](#)
[MXVR_DMA1_COUNT](#), [A-9](#)
[MXVR_DMA1_CURR_ADDR](#), [A-9](#)
[MXVR_DMA1_CURR_COUNT](#), [A-9](#)
[MXVR_DMA1_START_ADDR](#), [A-9](#)
[MXVR_DMA2_CONFIG](#), [A-9](#)
[MXVR_DMA2_COUNT](#), [A-9](#)
[MXVR_DMA2_CURR_ADDR](#), [A-9](#)
[MXVR_DMA2_CURR_COUNT](#), [A-9](#)
[MXVR_DMA2_START_ADDR](#), [A-9](#)
[MXVR_DMA3_CONFIG](#), [A-9](#)
[MXVR_DMA3_COUNT](#), [A-9](#)
[MXVR_DMA3_CURR_ADDR](#), [A-9](#)
[MXVR_DMA3_CURR_COUNT](#), [A-9](#)
[MXVR_DMA3_START_ADDR](#), [A-9](#)
[MXVR_DMA4_CONFIG](#), [A-9](#)
[MXVR_DMA4_COUNT](#), [A-10](#)
[MXVR_DMA4_CURR_ADDR](#), [A-10](#)
[MXVR_DMA4_CURR_COUNT](#),
[A-10](#)
[MXVR_DMA4_START_ADDR](#), [A-10](#)
[MXVR_DMA5_CONFIG](#), [A-10](#)
[MXVR_DMA5_COUNT](#), [A-10](#)
[MXVR_DMA5_CURR_ADDR](#), [A-10](#)
[MXVR_DMA5_CURR_COUNT](#),
[A-10](#)
[MXVR_DMA5_START_ADDR](#), [A-10](#)
[MXVR_DMA6_CONFIG](#), [A-10](#)
[MXVR_DMA6_COUNT](#), [A-10](#)
[MXVR_DMA6_CURR_ADDR](#), [A-10](#)
[MXVR_DMA6_CURR_COUNT](#),
[A-10](#)
[MXVR_DMA6_START_ADDR](#), [A-10](#)
[MXVR_DMA7_CONFIG](#), [A-10](#)
[MXVR_DMA7_COUNT](#), [A-11](#)
[MXVR_DMA7_CURR_ADDR](#), [A-11](#)
[MXVR_DMA7_CURR_COUNT](#),
[A-11](#)
[MXVR_DMA7_START_ADDR](#), [A-10](#)
[MXVR_FRAME_CNT_0](#), [A-12](#)
[MXVR_FRAME_CNT_1](#), [A-12](#)
[MXVR_GADDR](#), [A-7](#)
[MXVR_INT_EN_0](#), [A-7](#)
[MXVR_INT_EN_1](#), [A-7](#)
[MXVR_INT_STAT_0](#), [A-7](#)
[MXVR_INT_STAT_1](#), [A-7](#)
[MXVR_LADDR](#), [A-7](#)

Index

- MXVR_MAX_DELAY, [A-7](#)
- MXVR_MAX_POSITION, [A-7](#)
- MXVR_PAT_DATA_0, [A-11](#)
- MXVR_PAT_DATA_1, [A-11](#)
- MXVR_PAT_EN_0, [A-11](#)
- MXVR_PAT_EN_1, [A-11](#)
- MXVR_PLL_CTL, [A-7](#), [A-12](#)
- MXVR_POSITION, [A-7](#)
- MXVR_ROUTING_0, [A-12](#)
- MXVR_ROUTING_1, [A-12](#)
- MXVR_ROUTING_10, [A-12](#)
- MXVR_ROUTING_11, [A-12](#)
- MXVR_ROUTING_12, [A-12](#)
- MXVR_ROUTING_13, [A-12](#)
- MXVR_ROUTING_2, [A-12](#)
- MXVR_ROUTING_3, [A-12](#)
- MXVR_ROUTING_4, [A-12](#)
- MXVR_ROUTING_5, [A-12](#)
- MXVR_ROUTING_6, [A-12](#)
- MXVR_ROUTING_7, [A-12](#)
- MXVR_ROUTING_8, [A-12](#)
- MXVR_ROUTING_9, [A-12](#)
- MXVR_RRDB_CURR_ADDR, [A-11](#)
- MXVR_RRDB_START_ADDR, [A-11](#)
- MXVR_STATE_0, [A-7](#)
- MXVR_STATE_1, [A-7](#)
- MXVR_SYNC_LCHAN_0, [A-8](#)
- MXVR_SYNC_LCHAN_1, [A-8](#)
- MXVR_SYNC_LCHAN_2, [A-8](#)
- MXVR_SYNC_LCHAN_3, [A-8](#)
- MXVR_SYNC_LCHAN_4, [A-8](#)
- MXVR_SYNC_LCHAN_5, [A-8](#)
- MXVR_SYNC_LCHAN_6, [A-8](#)
- MXVR_SYNC_LCHAN_7, [A-8](#)
- registers
 - See also* registers by name
 - ATAPI, [24-48](#), [A-16](#)
 - diagram conventions, [-xlix](#)
 - system, [A-3](#)
- RegistersMXVR_ALLOC_6, [A-7](#)
- REG_RXBUFFER (device receive buffer)
 - bits, [24-55](#)
- REG_TXBUFFER (device transmit buffer)
 - bits, [24-54](#)
- remote frames, CAN, [27-22](#)
- Remote GetSource Control Message
 - Transmission, [21-147](#)
- Remote GetSource Control Message
 - Transmit Buffer Entry Field Offsets, [21-149](#)
- Remote Get Source In Progress (RGSIP) bit, [21-24](#)
- Remote GetSource Reception, [21-156](#)
- Remote Get Source system control
 - message, [21-24](#)
- Remote Read and Remote Write
 - Reception, [21-153](#)
- Remote Read Buffer, [21-15](#)
- Remote Read Buffer Field Offsets, [21-154](#)
- Remote Read Control Message
 - Transmission, [21-137](#)
- Remote Read Control Message Transmit
 - Buffer Entry Field Offsets, [21-138](#)
- Remote Read In Progress (RRDIP) bit, [21-25](#)
- Remote Write Complete interrupt enable, [21-46](#)
- Remote Write control message, [21-15](#)
- Remote Write Control Message Complete (RWRC) interrupt event, [21-38](#)
- Remote Write Control Message
 - Transmission, [21-139](#)
- Remote Write Control Message Transmit
 - Buffer Entry Field Offsets, [21-140](#)
- Remote Write In Progress (RWRIP) bit, [21-25](#)
- Remote Write Receive Enable (RWRRXEN) bit, [21-15](#)
- REQPKT (request an IN transaction) bit, [32-98](#)

- REQPKT_RH (request an IN transaction) bit, [32-109](#)
- request and IN transaction (STALL_RECEIVED_RH) bit, [32-109](#)
- request and IN transaction (STALL_RECEIVED_TH) bit, [32-102](#)
- request an IN transaction (REQPKT_RH) bit, [32-109](#)
- request an IN transaction (REQPKT) bit, [32-98](#)
- request an IN transaction (RXSTALL_TH) bit, [32-102](#)
- reset, effect on SPI, [28-21](#)
- Reset Asynchronous Packet Arbitration (RESETAP) bit, [21-78](#)
- reset endpoint data toggle (CLEAR_DATATOGGLE_R) bit, [32-109](#)
- reset endpoint data toggle (CLEAR_DATATOGGLE_T) bit, [32-102](#)
- RESET_OR_BABLE_BE (reset or bable IRQ enable) bit, [32-90](#)
- RESET_OR_BABLE_B (reset or bable indicator) bit, [32-89](#)
- reset or bable indicator (RESET_OR_BABLE_B) bit, [32-89](#)
- reset or bable IRQ enable (RESET_OR_BABLE_BE) bit, [32-90](#)
- RESET (USB reset) bit, [32-82](#)
- reset vector, [20-35](#)
- Resource Allocate Control Message Transmission, [21-141](#)
- Resource Allocate Control Message Transmit Buffer Entry Field Offsets, [21-142](#)
- Resource Allocate In Progress (ALIP) bit, [21-25](#)
- Resource Allocate Reception, [21-154](#)
- Resource De-Allocate Control Message Transmission, [21-144](#)
- Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets, [21-145](#)
- Resource De-Allocate In Progress (DALIP) bit, [21-24](#)
- Resource De-Allocate Reception, [21-155](#)
- RESP_CMD (response command) bits, [23-25](#)
- response (CMD_RSP) bit, [23-24](#)
- response command (RESP_CMD) bits, [23-25](#)
- RESUME_BE (resume signalling IRQ enable) bit, [32-90](#)
- RESUME_B (resume signalling indicator) bit, [32-89](#)
- resume mode flag (RESUME_MODE) bit, [32-82](#)
- RESUME_MODE (resume mode flag) bit, [32-82](#)
- resume signalling indicator (RESUME_B) bit, [32-89](#)
- resume signalling IRQ enable (RESUME_BE) bit, [32-90](#)
- re-synchronization, CAN, [27-12](#)
- RFCS (receive FIFO count status) bit, [31-39](#)
- RFHn bit, [27-83](#)
- RFIT (receive FIFO IRQ threshold) bit, [31-33](#)
- RFRT (receive FIFO RTS threshold) bit, [31-33](#)
- RFS pins, [30-20](#), [30-32](#)
- RFSR bit, [30-32](#), [30-33](#), [30-57](#), [30-60](#)
- RFSx signal, [30-6](#)

Index

- RGB_FMT_EN (formatting enable) bit, [26-83](#)
- RLSBIT bit, [30-57](#), [30-59](#)
- RMLIF bit, [27-27](#), [27-52](#)
- RMLIM bit, [27-27](#), [27-50](#)
- RMLIS bit, [27-27](#), [27-51](#)
- RMLn bit, [27-76](#)
- RMPn bit, [27-75](#)
- rod control (ROD_CTL) bit, [23-22](#)
- ROD_CTL (rod control) bit, [23-22](#)
- route Rx IRQ to INTx (RX_INTx_R) bits, [32-94](#)
- route Tx IRQ to INTx (TX_INTx_R) bits, [32-94](#)
- route USB/VBUS IRQ to INTx (USB_INTx_R) bits, [32-94](#)
- ROVF bit, [30-65](#), [30-66](#), [30-67](#)
- row enable width (KPAD_ROWEN) bits, [22-10](#)
- rows value pressed (KPAD_ROW) bits, [22-16](#)
- RPOLC bit, [31-50](#), [31-51](#)
- RRFST bit, [30-58](#), [30-60](#)
- RSCLKx pins, [30-31](#)
- RSCLKx signal, [30-6](#)
- RSFSE bit, [30-13](#), [30-58](#), [30-60](#)
- RSPEN bit, [30-11](#), [30-56](#), [30-57](#), [30-58](#)
- RSTART bit, [29-41](#), [29-42](#)
- RTC, [20-30](#)
- RTR bit, [27-57](#)
- RUVF bit, [30-65](#), [30-66](#), [30-67](#)
- RWRCEN, [21-46](#)
- RWR (read wait request) bit, [23-38](#)
- RX_ACT_MASK (receive active mask) bit, [23-33](#)
- RX_ACT (receive active) bit, [23-30](#)
- RX_COUNT (USB Rx byte count) bits, [32-116](#)
- Rx data buffer status (RXS) bit, [28-48](#)
- RX_DAT_RDY_MASK (receive data available mask) bit, [23-33](#)
- RX_DAT_RDY (receive data available) bit, [23-31](#)
- RXECNT[7:0] field, [27-89](#)
- RX_FIFO_FULL_MASK (receive FIFO full mask) bit, [23-33](#)
- RX_FIFO_FULL (receive FIFO full) bit, [23-31](#)
- RX_FIFO_STAT_MASK (receive FIFO status mask) bit, [23-33](#)
- RX_FIFO_STAT (receive FIFO status) bit, [23-30](#)
- RX_FIFO_ZERO_MASK (receive FIFO empty mask) bit, [23-33](#)
- RX_FIFO_ZERO (receive FIFO empty) bit, [23-31](#)
- RX hold register, [30-64](#)
- RX_INTx_R (route Rx IRQ to INTx) bits, [32-94](#)
- RXNE bit, [30-67](#)
- RX_OVERRUN (receive overrun) bit, [23-30](#)
- Rx packet serviced (SERVICED_RXPKTRDY) bit, [32-98](#)
- RxPktRdy autoclear enable (AUTOCLEAR_R) bit, [32-109](#)
- RXPKTRDY (data packet receive indicator) bit, [32-98](#)
- RXPKTRDY_R (data packet in FIFO indicator) bit, [32-109](#)
- RX_POLL_INTERVAL (USB Rx poll interval) bits, [32-120](#)
- Rx protocol type (PROTOCOL_R) bits, [32-119](#)
- RXREQ signal, [31-9](#)
- RXS bit, [28-31](#), [28-48](#)
- RXSE bit, [30-58](#), [30-60](#)
- RXS (Rx data buffer status) bit, [28-48](#)

RXSTALL_TH (request an IN transaction)
bit, [32-102](#)

RX_UNDERRUN_MASK (receive
underrun mask) bit, [23-33](#)

RX_UNDERRUN_STAT (receive
underrun status) bit, [23-32](#)

S

SA0 bit, [27-89](#)

SADDR[6:0] field, [29-39](#)

SAM bit, [27-49](#)

SAMPLE/PRELOAD instruction, [B-7](#)

sampling, CAN, [27-12](#)

sampling edge, SPORT, [30-35](#)

SBL2UEN, [21-45](#)

SB (set break) bit, [31-30](#)

SBU2LEN, [21-45](#)

SBUEN, [21-45](#)

scan paths, [B-5](#)

SCCB bit, [29-36](#)

SCD_MSK (mask card detect) bit, [23-36](#)

SCK signal, [28-5](#), [28-17](#), [28-19](#), [28-22](#)

SCL clock divider (TWI_CLKDIV)
register, [29-36](#)

SCLK

SCLK domain, [20-11](#)

SCLOVR bit, [29-41](#)

SCL pin, [29-5](#)

SCLSEN bit, [29-13](#), [29-45](#)

SCOMP bit, [29-21](#), [29-51](#)

SCOMPM bit, [29-48](#), [29-51](#)

scratch[7:0] field, [31-49](#)

SCTS (sticky CTS) bit, [31-39](#)

SD4E (SDIO 4-bit enable) bit, [23-37](#)

SDAOVR bit, [29-41](#), [29-42](#)

SDA pin, [29-5](#)

SDASEN bit, [29-14](#), [29-45](#)

SD card detect (SD_CARD_DET) bit,
[23-35](#)

SD_CARD_DET (SD card detect) bit,
[23-35](#)

SD_CLK bus clock enable (CLK_E) bit,
[23-23](#)

SD_CLK divisor (CLKDIV) bits, [23-23](#)

SD_CMD_OD (open drain output) bit,
[23-22](#)

SDH_ARGUMENT (SDH argument)
register, [23-20](#), [23-24](#), [A-14](#)

SDH argument (SDH_ARGUMENT)
register, [23-20](#), [23-24](#), [A-14](#)

SDH_CFG (SDH configuration) register,
[23-21](#), [23-37](#), [A-15](#)

SDH_CLK_CTL (SDH clock control)
register, [23-19](#), [23-23](#), [A-14](#)

SDH clock control (SDH_CLK_CTL)
register, [23-19](#), [23-23](#), [A-14](#)

SDH_COMMAND (SDH command)
register, [23-20](#), [23-24](#), [A-14](#)

SDH command (SDH_COMMAND)
register, [23-20](#), [23-24](#), [A-14](#)

SDH configuration (SDH_CFG) register,
[23-21](#), [23-37](#), [A-15](#)

SDH_DATA_CNT (SDH data counter)
register, [23-20](#), [23-29](#), [A-14](#)

SDH data control (SDH_DATA_CTL)
register, [23-20](#), [23-27](#), [A-14](#)

SDH data counter (SDH_DATA_CNT)
register, [23-20](#), [23-29](#), [A-14](#)

SDH_DATA_CTL (SDH data control)
register, [23-20](#), [23-27](#), [A-14](#)

SDH data FIFO (SDH_FIFOx) registers,
[23-21](#), [23-35](#), [A-15](#)

SDH data length (SDH_DATA_LGTH)
register, [23-20](#), [23-27](#), [A-14](#)

SDH_DATA_LGTH (SDH data length
register, [23-20](#), [23-27](#), [A-14](#)

SDH_DATA_TIMER (SDH data timer)
register, [23-20](#), [23-26](#), [A-14](#)

Index

- SDH data timer (SDH_DATA_TIMER)
 - register, [23-20](#), [23-26](#), [A-14](#)
- SDH_E_MASK (SDH exception mask)
 - register, [23-21](#), [23-36](#), [A-15](#)
- SDH_E_STATUS (SDH exception status)
 - register, [23-21](#), [23-35](#), [A-15](#)
- SDH exception mask (SDH_E_MASK)
 - register, [23-21](#), [23-36](#), [A-15](#)
- SDH exception status (SDH_E_STATUS)
 - register, [23-21](#), [23-35](#), [A-15](#)
- (SDH_FIFO_CNT (SDH FIFO counter)
 - register, [23-34](#)
- SDH_FIFO_CNT (SDH FIFO counter)
 - register, [23-21](#), [A-15](#)
- SDH FIFO counter (SDH_FIFO_CNT)
 - register, [23-21](#), [23-34](#), [A-15](#)
- SDH_FIFOx (SDH data FIFO) registers,
 - [23-21](#), [23-35](#), [A-15](#)
- SDH identification (SDH_PIDx) registers,
 - [23-21](#), [23-38](#), [A-15](#)
- SDH interrupt mask (SDH_MASKx)
 - registers, [23-20](#), [23-33](#), [A-15](#)
- SDH_MASKx (SDH interrupt mask)
 - registers, [23-20](#), [23-33](#), [A-15](#)
- SDH_PIDx (SDH identification) registers,
 - [23-21](#), [23-38](#), [A-15](#)
- SDH power control (SDH_PWR_CTL)
 - register, [23-19](#), [23-22](#), [A-14](#)
- SDH_PWR_CTL (SDH power control)
 - register, [23-19](#), [23-22](#), [A-14](#)
- SDH_RD_WAIT_EN (SDH read wait enable) register, [23-21](#), [23-38](#), [A-15](#)
- SDH read wait enable
 - (SDH_RD_WAIT_EN) register,
 - [23-21](#), [23-38](#), [A-15](#)
- SDH_RESP_CMD (SDH response command) register, [23-20](#), [23-25](#), [A-14](#)
- SDH response command
 - (SDH_RESP_CMD) register, [23-20](#), [23-25](#), [A-14](#)
- SDH response (SDH_RESPONSEx)
 - registers, [23-20](#), [23-26](#), [A-14](#)
- SDH_RESPONSEx (SDH response)
 - registers, [23-20](#), [23-26](#), [A-14](#)
- SDH status clear (SDH_STATUS_CLR)
 - register, [23-20](#), [23-32](#), [A-15](#)
- SDH_STATUS_CLR (SDH status clear)
 - register, [23-20](#), [23-32](#), [A-15](#)
- SDH_STATUS (SDH status) register,
 - [23-20](#), [23-30](#), [A-15](#)
- SDH status (SDH_STATUS) register,
 - [23-20](#), [23-30](#), [A-15](#)
- SDIO 4-bit enable (SD4E) bit, [23-37](#)
- SDIO_INT_DET (SDIO interrupt detected) bit, [23-35](#)
- SDIO interrupt detected
 - (SDIO_INT_DET) bit, [23-35](#)
- SDIO interrupt detected (SDIO_MSK)
 - bit, [23-36](#)
- SDIO_MSK (SDIO interrupt detected)
 - bit, [23-36](#)
- SDIR bit, [29-16](#), [29-40](#)
- SDMMC reset (SD_RST) bit, [23-37](#)
- SD_RST (SDMMC reset) bit, [23-37](#)
- secure digital host
 - descripition of operation, [23-3](#)
 - MMC/SD card detection, [23-18](#)
 - receive FIFO, [23-17](#)
- SDH clocking, [23-4](#)
- SDH data, [23-10](#)
- SDH data FIFO, [23-16](#)
- SDH data path state machine, [23-10](#)
- SDH operation, [23-5](#)
- SDH registers, [23-19](#), [A-13](#)
 - SDH_CLK_CTL, [23-19](#), [A-14](#)

- SDH_PWR_CTL, [23-19](#), [A-14](#)
- SDIO interrupt and read wait support, [23-17](#)
 - transmit FIFO, [23-16](#)
- select cycle time - TDVS time (TCYC_TDVS) bits, [24-68](#)
- select data valid setup time (TDVS) bits, [24-68](#)
- select DIOR/DIOW pulsewidth (TEOC_REG) bits, [24-64](#)
- select DIOR negated pulsewidth (TKR) bits, [24-66](#)
- select DIOW data hold (TH) bits, [24-67](#)
- select DIOW negated pulsewidth (TKW) bits, [24-66](#)
- SELECTED_ENDPOINT (USB endpoint index) bits, [32-91](#), [32-93](#)
- select end of cycle for DMA (TEOC) bits, [24-67](#)
- select envelope time (TENV) bits, [24-67](#)
- select interlock time (TMLI) bits, [24-68](#)
- select minimum delay required for output (TZAH) bits, [24-69](#)
- select ready to pause (READY_PAUSE) bits, [24-69](#)
- select setup and hold times for TACK (TACK) bits, [24-67](#)
- select time from STROBE edge to negation of DMARQ or assertion of STOP (TSS) bits, [24-68](#)
- SEN bit, [29-37](#), [29-38](#)
- send setup token (SETUPPKT) bit, [32-98](#)
- send STALL handshake (SENDSTALL) bit, [32-98](#)
- send STALL handshake (STALL_SEND_R) bit, [32-109](#)
- send STALL handshake (STALL_SEND_T) bit, [32-102](#)
- SENDSTALL (send STALL handshake) bit, [32-98](#)
- send zero (SZ) bit, [28-45](#)
- SER bit, [27-89](#)
- serial clock divide modulus[15:0] field, [30-68](#)
- serial clock frequency, [28-22](#)
- serial communications, [31-6](#)
- serial data transfer, [30-3](#)
- serial scan paths, [B-5](#)
- SERR bit, [29-20](#), [29-51](#)
- SERRM bit, [29-48](#), [29-50](#)
- SERVICED_RXPKTRDY (Rx packet serviced) bit, [32-98](#)
- SERVICED_SETUPEND (setup end serviced) bit, [32-98](#)
- session end/disconnect indicator (DISCON_B) bit, [32-89](#)
- session end/disconnect IRQ enable (DISCON_BE) bit, [32-90](#)
- session indicator (SESSION) bit, [32-122](#), [32-124](#)
- SESSION_REQ_BE (session request IRQ enable) bit, [32-90](#)
- SESSION_REQ_B (session request indicator) bit, [32-89](#)
- session request indicator (SESSION_REQ_B) bit, [32-89](#)
- session request IRQ enable (SESSION_REQ_BE) bit, [32-90](#)
- SESSION (session indicator) bit, [32-122](#), [32-124](#)
- setup end serviced (SERVICED_SETUPEND) bit, [32-98](#)
- SETUPEND (setup end) bit, [32-98](#)
- setup end (SETUPEND) bit, [32-98](#)
- SETUPPKT (send setup token) bit, [32-98](#)
- shorten startup counter chain (TM_SHORT_CHAIN) bit, [32-132](#)
- SIGN_EXT (sign extension/zero-filled) bit, [26-83](#)

Index

- sine wave input, [20-32](#)
- Single cast Transmission Status Encodings, [21-133](#)
- single shot transmission, CAN, [27-15](#)
- SINIT bit, [29-21](#), [29-51](#)
- SINITM bit, [29-48](#), [29-51](#)
- SIZE bit, [28-21](#), [28-45](#)
- size of words (SIZE) bit, [28-45](#)
- SIZE (size of words) bit, [28-45](#)
- SJW[1:0] field, [27-12](#), [27-49](#)
- SKIP_EN (skip enable) bit, [26-83](#)
- SKIP_EO (skip even/odd) bit, [26-83](#)
- Slave Mode, [21-13](#)
- Slave mode initialization, [21-118](#)
- slave select, SPI, [28-47](#)
- slave select enable (FLSx) bits, [28-46](#)
- slave select enable (PSSE) bit, [28-45](#)
- slave SPI device, [28-7](#)
- sleep mode, [20-33](#)
 - CAN, [27-40](#)
- SLEN[4:0] field, [30-52](#), [30-54](#), [30-58](#), [30-59](#)
 - restrictions, [30-30](#)
 - word length formula, [30-30](#)
- SMODE_B (switch charge pump mode) bits, [32-132](#)
- SMR bit, [27-46](#)
- SOF_BE (start of frame IRQ enable) bit, [32-90](#)
- SOF_B (start of frame indicator) bit, [32-89](#)
- soft connect enable (SOFTC_CONN) bit, [32-82](#)
- SOFT_CONN (soft connect enable) bit, [32-82](#)
- SOFT_RST (soft reset) bit, [24-50](#)
- software programmable force evaluate (KPAD_SOFTEVAL_E) bit, [22-21](#)
- software reset, and CAN, [27-13](#)
- software reset (SRS) bit, [27-46](#)
- software watchdog timer, [20-31](#)
- SOVF bit, [29-20](#), [29-51](#)
- SOVFM bit, [29-48](#), [29-50](#)
- SPE bit, [28-22](#), [28-45](#)
- SPE (SPI enable) bit, [28-45](#)
- SPI, [20-22](#), [28-1](#) to [28-58](#)
 - beginning and ending transfers, [28-30](#)
 - bit mapping to port pins, [28-10](#)
 - block diagram, [28-3](#)
 - clock phase, [28-17](#), [28-19](#), [28-22](#)
 - clock polarity, [28-17](#), [28-22](#)
 - clock signal, [28-3](#), [28-22](#)
 - code examples, [28-50](#)
 - data corruption, avoiding, [28-19](#)
 - data interrupt, [28-25](#)
 - data transfer, [28-20](#)
 - detecting transfer complete, [28-23](#) and DMA, [28-14](#)
 - DMA initialization, [28-54](#)
 - DMA transfers, [28-53](#)
 - effect of reset, [28-21](#)
 - error interrupt, [28-25](#)
 - error signals, [28-23](#) to [28-25](#)
 - general operation, [28-26](#) to [28-30](#)
 - initialization, [28-50](#)
 - internal interfaces, [28-14](#)
 - interrupt outputs, [28-25](#)
 - interrupts, [28-52](#)
 - master mode, [28-20](#), [28-26](#)
 - master mode DMA operation, [28-33](#)
 - mode fault error, [28-24](#)
 - multiple slave systems, [28-12](#)
 - port F, [28-4](#)
 - reception error, [28-25](#)
 - registers, table, [28-43](#)
 - SCK signal, [28-5](#)
 - slave device, [28-7](#)
 - slave mode, [28-20](#), [28-29](#)
 - slave mode DMA operation, [28-35](#)
 - slave select enable setup, [28-2](#), [28-9](#)
 - slave-select function, [28-46](#)

- slave transfer preparation, [28-30](#)
- SPI_FLG mapping to port pins, [28-47](#)
- starting DMA transfer, [28-56](#)
- starting transfer, [28-51](#)
- stopping, [28-53](#)
- stopping DMA transfers, [28-56](#)
- switching between transmit and receive, [28-32](#)
- timing, [28-8](#)
- transfer formats, [28-17](#) to [28-19](#)
- transfer initiate command, [28-27](#)
- transfer modes, [28-28](#)
- transfer protocol, [28-18](#), [28-19](#)
- transmission error, [28-25](#)
- transmission/reception errors, [28-23](#)
- transmit collision error, [28-25](#)
- using DMA, [28-14](#)
- word length, [28-21](#)
- SPI baud rate registers (SPI_BAUD), [28-22](#), [28-43](#)
- SPI baud rate (SPIx_BAUD) registers, [28-44](#)
- SPI_BAUD (SPI baud rate registers), [28-22](#), [28-43](#)
- SPI_BAUD values, [28-23](#)
- SPI control register (SPI_CTL), [28-21](#), [28-43](#), [28-45](#)
- SPI control (SPIx_CTL) registers, [28-45](#)
- SPI_CTL (SPI control register), [28-5](#), [28-21](#), [28-43](#), [28-45](#)
- SPI enable (SPE) bit, [28-45](#)
- SPIF bit, [28-13](#), [28-31](#), [28-48](#)
- SPI finished (SPIF) bit, [28-48](#)
- SPI flag register (SPI_FLG), [28-10](#), [28-43](#), [28-46](#)
- SPI flag (SPIx_FLG) registers, [28-46](#)
- SPI_FLG bit, [28-10](#)
- SPI_FLG (SPI flag register), [28-10](#), [28-12](#), [28-43](#), [28-46](#)
- SPIF (SPI finished) bit, [28-48](#)
- SPI_RDBR shadow[15:0] field, [28-49](#)
- SPI RDBR shadow register (SPI_SHADOW), [28-16](#), [28-44](#), [28-49](#)
- SPI RDBR shadow (SPIx_SHADOW) registers, [28-49](#)
- SPI_RDBR (SPI receive data buffer register), [28-16](#), [28-44](#), [28-49](#)
- SPI receive data buffer register (SPI_RDBR), [28-16](#), [28-44](#), [28-49](#)
- SPI receive data buffer (SPIx_RDBR) registers, [28-49](#)
- SPI_SHADOW (SPI RDBR shadow register), [28-16](#), [28-44](#), [28-49](#)
- SPI slave select, [28-47](#)
- SPISS signal, [28-7](#), [28-12](#), [28-17](#)
- SPI_STAT (SPI status register), [28-23](#), [28-43](#), [28-48](#)
- SPI status register (SPI_STAT), [28-23](#), [28-43](#), [28-48](#)
- SPI status (SPIx_STAT) registers, [28-48](#)
- SPI_TDBR (SPI transmit data buffer register), [28-15](#), [28-43](#), [28-48](#)
- SPI transmit data buffer register (SPI_TDBR), [28-15](#), [28-43](#), [28-48](#)
- SPI transmit data buffer (SPIx_TDBR) registers, [28-48](#)
- SPIx_BAUD (SPI baud rate) registers, [28-44](#)
- SPIx_CTL (SPI control) registers, [28-45](#)
- SPIx_FLG (SPI flag) registers, [28-46](#)
- SPIx_RDBR (SPI receive data buffer) registers, [28-49](#)
- SPIx_SHADOW (SPI RDBR shadow) registers, [28-49](#)
- SPIx_STAT (SPI status) registers, [28-48](#)
- SPIx_TDBR data buffer status (TXS) bit, [28-48](#)
- SPIx_TDBR (SPI transmit data buffer) registers, [28-48](#)

Index

- SPLT_EVEN_ODD, 26-83
- SPORT, 20-20, 30-1 to 30-82
 - active low vs. active high frame syncs, 30-35
 - channels, 30-17
 - clock, 30-31
 - clock frequency, 30-28, 30-68
 - clock rate, 30-2
 - clock rate restrictions, 30-29
 - clock recovery control, 30-27
 - companding, 30-31
 - configuration, 30-12
 - data formats, 30-30
 - data word formats, 30-61
 - delay when enabled, 30-12
 - disabling, 30-12
 - DMA data packing, 30-26
 - enable/disable, 30-11
 - enabling multichannel mode, 30-19
 - framed serial transfers, 30-33
 - framed vs. unframed, 30-32
 - frame sync, 30-34, 30-37
 - frame sync frequencies, 30-28
 - framing signals, 30-32
 - general operation, 30-11
 - H.100 standard protocol, 30-27
 - initialization code, 30-59
 - internal memory access, 30-40
 - internal vs. external frame syncs, 30-34
 - late frame sync, 30-19
 - modes, 30-12
 - moving data to memory, 30-40
 - multichannel frame, 30-22
 - multichannel operation, 30-17 to 30-26
 - multichannel transfer timing, 30-18
 - multiplexed pins, 30-4
 - PAB error, 30-41
 - packing data, multichannel DMA, 30-26
 - pins, 30-4
 - port connection, 30-8
 - port G, 30-4
 - receive and transmit functions, 30-4
 - receive clock signal, 30-31
 - receive FIFO, 30-63
 - receive word length, 30-64
 - register writes, 30-50
 - RX hold register, 30-64
 - sampling edge, 30-35
 - selecting bit order, 30-30
 - serial data communication protocols, 30-1
 - shortened active pulses, 30-12
 - signals, 30-5
 - single clock for both receive and transmit, 30-31
 - single word transfers, 30-40
 - stereo serial connection, 30-10
 - stereo serial frame sync modes, 30-19
 - stereo serial operation, 30-13
 - support for standard protocols, 30-27
 - termination, 30-10
 - timing, 30-41
 - transmit clock signal, 30-31
 - transmitter FIFO, 30-61
 - transmit word length, 30-62
 - TX hold register, 30-62
 - TX interrupt, 30-62
 - unframed data flow, 30-33
 - unpacking data, multichannel DMA, 30-26
 - window offset, 30-24
 - word length, 30-29
- SPORT current channel
 - (SPORTx_CHNL) registers, 30-49, 30-71
- SPORT error interrupt, 30-40
- SPORT multichannel configuration
 - (SPORTx_MCMC1) register 1, 30-49

- SPORT multichannel configuration (SPORT_x_MCMC2) register 2, [30-49](#)
- SPORT multichannel receive select (SPORT_x_MRCSn) registers, [30-49](#)
- SPORT multichannel receive select (SPORT_x_MRCSn) registers, [30-72](#)
- SPORT multichannel transmit select registers (SPORT_x_MTCSn), [30-25](#)
- SPORT multichannel transmit select (SPORT_x_MTCSn) registers, [30-49](#), [30-74](#)
- SPORT receive configuration 1 (SPORT_x_RCR1) registers, [30-49](#), [30-56](#)
- SPORT receive configuration 2 (SPORT_x_RCR2) registers, [30-49](#), [30-56](#)
- SPORT receive data (SPORT_x_RX) registers, [30-49](#), [30-63](#), [30-65](#)
- SPORT receive frame sync divider (SPORT_x_RFSDIV) registers, [30-49](#), [30-69](#)
- SPORT receive serial clock divider (SPORT_x_RCLKDIV) registers, [30-49](#), [30-68](#)
- SPORT RX interrupt, [30-40](#), [30-65](#)
- SPORT status (SPORT_x_STAT) registers, [30-49](#), [30-66](#)
- SPORT transmit configuration 1 (SPORT_x_TCR1) registers, [30-48](#), [30-51](#)
- SPORT transmit configuration 2 (SPORT_x_TCR2) registers, [30-48](#), [30-51](#)
- SPORT transmit data (SPORT_x_TX) registers, [30-48](#), [30-61](#), [30-62](#)
- SPORT transmit frame sync divider (SPORT_x_TFSDIV) registers, [30-48](#), [30-69](#)
- SPORT transmit serial clock divider (SPORT_x_TCLKDIV) registers, [30-48](#), [30-68](#)
- SPORT TX interrupt, [30-40](#)
- SPORT_x_CHNL (SPORT current channel) registers, [30-49](#), [30-71](#)
- SPORT_x_MCMC1 (SPORT multichannel configuration) register 1, [30-49](#)
- SPORT_x_MCMC2 (SPORT multichannel configuration) register 2, [30-49](#)
- SPORT_x_MRCSn (SPORT multichannel receive select) registers, [30-49](#)
- SPORT_x_MRCSn (SPORT multichannel receive select) registers, [30-72](#)
- SPORT_x_MTCSn (SPORT multichannel transmit select) registers, [30-49](#), [30-74](#)
- SPORT_x multichannel configuration registers (SPORT_x_MCMCn), [30-70](#)
- SPORT_x multichannel receive select registers (SPORT_x_MRCSn), [30-25](#)
- SPORT_x multichannel transmit select registers (SPORT_x_MTCSn), [30-25](#)
- SPORT_x_RCLKDIV (SPORT receive serial clock divider) registers, [30-49](#), [30-68](#)
- SPORT_x_RCR1 (SPORT receive configuration 1) registers, [30-49](#), [30-56](#)
- SPORT_x_RCR2 (SPORT receive configuration 2) registers, [30-49](#), [30-56](#)
- SPORT_x_RCR2 (SPORT_x receive configuration register), [30-58](#)
- SPORT_x receive configuration 2 registers (SPORT_x_RCR2), [30-58](#)
- SPORT_x receive data registers (SPORT_x_RX), [30-21](#)

Index

- SPORT_x_RFSDIV (SPORT receive frame sync divider) registers, [30-49](#), [30-69](#)
- SPORT_x_RX (SPORT receive data) registers, [30-49](#), [30-63](#), [30-65](#)
- SPORT_x_STAT (SPORT status) registers, [30-49](#), [30-66](#)
- SPORT_x_TCLKDIV (SPORT transmit serial clock divider) registers, [30-48](#), [30-68](#)
- SPORT_x_TCR1 (SPORT transmit configuration 1) registers, [30-48](#), [30-51](#)
- SPORT_x_TCR2 (SPORT transmit configuration 2) registers, [30-48](#), [30-51](#)
- SPORT_x_TFSDIV (SPORT transmit frame sync divider) registers, [30-48](#), [30-69](#)
- SPORT_x transmit data registers (SPORT_x_TX), [30-21](#), [30-39](#)
- SPORT_x_TX (SPORT transmit data) registers, [30-48](#), [30-61](#), [30-62](#)
- SRS bit, [27-46](#)
- SRS (software reset) bit, [27-46](#)
- STALL handshake received (STALL_RECEIVED) bit, [32-98](#)
- STALL handshake sent (STALL_SENT) bit, [32-98](#)
- STALL handshake sent (STALL_SENT_R) bit, [32-109](#)
- STALL handshake sent (STALL_SENT_T) bit, [32-102](#)
- STALL_RECEIVED_RH (request and IN transaction) bit, [32-109](#)
- STALL_RECEIVED (STALL handshake received) bit, [32-98](#)
- STALL_RECEIVED_TH (request and IN transaction) bit, [32-102](#)
- STALL_SEND_R (send STALL handshake) bit, [32-109](#)
- STALL_SEND_T (send STALL handshake) bit, [32-102](#)
- STALL_SENT_R (STALL handshake sent) bit, [32-109](#)
- STALL_SENT (STALL handshake sent) bit, [32-98](#)
- STALL_SENT_T (STALL handshake sent) bit, [32-102](#)
- Start Asynchronous Packet Transmission (STARTAP) bit, [21-77](#)
- START_BIT_ERR bit, [23-30](#)
- START_BIT_ERR_MASK bit, [23-33](#)
- start bit error status (START_BIT_ERR_STAT) bit, [23-32](#)
- START_BIT_ERR_STAT (start bit error status) bit, [23-32](#)
- Start Control Message Transmission (STARTCM) bit, [21-84](#)
- start of frame indicator (SOF_B) bit, [32-89](#)
- start of frame IRQ enable (SOF_B) bit, [32-90](#)
- Start Pattern select (STARTPAT_x) field, [21-67](#)
- Status Change Interrupt, [21-30](#), [21-31](#)
- STATUSPKT_H (packet transaction status) bit, [32-98](#)
- STB (stop bits) bit, [31-30](#)
- STDVAL bit, [29-37](#), [29-38](#)
- stereo serial data, [30-2](#)
- stereo serial device, SPORT connection, [30-10](#)
- stereo serial frame sync modes, [30-19](#)
- stereo serial operation, SPORT, [30-13](#)
- STOP bit, [29-43](#)
- Stop Mode, [21-63](#), [21-74](#)
- Stop Pattern select (STOPPAT_x) field, [21-68](#)
- STP (stick parity) bit, [31-30](#)

- SUBSPLT_ODD (sub-split odd samples) bit, [26-83](#)
 - Super Block Locked State (SBLOCK) bit, [21-31](#)
 - Super Block Locked to Unlocked interrupt enable, [21-45](#)
 - Super Block Locked to Unlocked (SBL2U) interrupt event, [21-31](#)
 - Super Block Lock (SBLOCK) bit, [21-20](#)
 - Super Block Unlocked to Locked interrupt enable, [21-45](#)
 - Super Block Unlocked to Locked (SBU2L) interrupt event, [21-31](#)
 - support, technical or customer, [-xlv](#)
 - SUSPEND_BE (suspend signalling IRQ enable) bit, [32-90](#)
 - SUSPEND_B (suspend signalling indicator) bit, [32-89](#)
 - suspend mode, CAN, [27-39](#)
 - suspend mode enable (SUSPEND_MODE) bit, [32-82](#)
 - suspend mode output enable (ENABLE_SUSPENDM) bit, [32-82](#)
 - SUSPEND_MODE (suspend mode enable) bit, [32-82](#)
 - suspend signalling indicator (SUSPEND_B) bit, [32-89](#)
 - suspend signalling IRQ enable (SUSPEND_BE) bit, [32-90](#)
 - SWAPEN (swap enable) bit, [26-83](#)
 - switch charge pump mode (SMODE_B) bits, [32-132](#)
 - SYNC bit, [31-26](#)
 - Synchronous Boundary (MSB) field, [21-17](#)
 - Synchronous Boundary Updated interrupt enable, [21-45](#)
 - Synchronous Boundary Updated (SBU) interrupt event, [21-33](#)
 - Synchronous Data Delay (SDELAY) bit, [21-13](#)
 - Synchronous Data Interrupt, [21-41](#)
 - Synchronous Data Reception, [21-126](#)
 - Synchronous Data Routing, Muting, and Transmission, [21-123](#)
 - Synchronous Packet Autobuffer Modes, [21-64](#)
 - Synchronous Packet-Fixed Count Mode, [21-71](#), [21-75](#), [21-76](#)
 - Synchronous Packet-Start/Stop Mode, [21-65](#), [21-71](#), [21-75](#), [21-77](#)
 - Synchronous Packet-Variable Count Mode, [21-64](#), [21-65](#), [21-71](#), [21-75](#), [21-76](#)
 - Synchronous Receive FIFO Number of Bytes (SRXNUMB) field, [21-27](#)
 - Synchronous Receive FIFO Number of Bytes (STXNUMB) field, [21-27](#)
 - synchronous serial data transfer, [30-3](#)
 - synchronous serial ports, *See* SPORT
 - system peripheral clock, *See* SCLK
 - system peripherals, [20-3](#)
 - system reset, [20-35](#) to ??
 - SZ bit, [28-29](#), [28-45](#)
 - SZ (send zero) bit, [28-45](#)
- ## T
- T1_REG (time from address valid to DIOR/DIOW) bits, [24-65](#)
 - T2_REG (end of cycle time for register access transfers) bits, [24-64](#)
 - T2_REG_PIO (DIOR/DIOW pulsewidth) bits, [24-65](#)
 - T4_REG (DIOW data hold) bits, [24-65](#)
 - TACK (select setup and hold times for TACK) bits, [24-67](#)
 - TAn bit, [27-81](#)
 - TAP registers
 - boundary-scan, [B-7](#)
 - bypass, [B-7](#)
 - instruction, [B-3](#), [B-5](#)

Index

- TAP (test access port), [B-2](#), [B-3](#)
 - controller, [B-3](#)
- TARGET_EP_NO_R (target EPx number) bits, [32-119](#)
- TARGET_EP_NO_T (target EPx number) bits, [32-117](#)
- target EPx number
 - (TARGET_EP_NO_R) bits, [32-119](#)
- target EPx number
 - (TARGET_EP_NO_T) bits, [32-117](#)
- TCKFE bit, [30-35](#), [30-51](#), [30-55](#)
- TCYC_TDVS (select cycle time - TDVS time) bits, [24-68](#)
- TDA bit, [27-24](#), [27-82](#)
- TD (DIOR/DIOW asserted pulsewidth) bits, [24-66](#)
- TDM interfaces, [30-3](#)
- TDPTR[4:0] field, [27-82](#)
- TDR bit, [27-24](#), [27-82](#)
- TDTYPE[1:0] field, [30-30](#), [30-51](#), [30-53](#)
- TDVS (select data valid setup time) bits, [24-68](#)
- technical support, [-xlv](#)
- TEMT bit, [31-8](#), [31-36](#), [31-38](#)
- TENV (select envelope time) bits, [24-67](#)
- TEOC_REG_PIO (end of cycle time for PIO access transfers) bits, [24-65](#)
- TEOC_REG (select DIOR/DIOW pulsewidth) bits, [24-64](#)
- TEOC (select end of cycle for DMA) bits, [24-67](#)
- terminations, SPORT pin/line, [30-10](#)
- test access port (TAP), [B-2](#), [B-3](#)
 - controller, [B-3](#)
- test clock (TCK), [B-7](#)
- test features, [B-1](#) to [B-7](#)
- testing circuit boards, [B-1](#), [B-5](#)
- test-logic-reset state, [B-4](#)
- TFI (transmission finished indicator) bit, [31-36](#), [31-38](#)
- TFRcnt_RST (transmission count reset) bit, [24-50](#)
- TFS pins, [30-32](#), [30-39](#)
- TFSR bit, [30-32](#), [30-33](#), [30-51](#), [30-54](#)
- TFS signal, [30-21](#)
- TFSx signal, [30-5](#)
- THRE bit, [31-18](#), [31-38](#)
- THRE flag, [31-8](#), [31-24](#)
- THRE (transmit hold register empty) bit, [31-36](#)
- throughput
 - SPORT, [30-8](#)
- TH (select DIOW data hold) bits, [24-67](#)
- time-division-multiplexed (TDM) mode, [30-17](#)
 - See also* SPORT, multichannel operation time from address valid to DIOR/DIOW (T1_REG) bits, [24-65](#)
 - time from address valid to DIOR/DIOW (TM) bits, [24-66](#)
- timeout error (ERROR_H) bit, [32-98](#)
- timeout error indicator (ERROR_RH) bit, [32-109](#)
- timeout error indicator (ERROR_TH) bit, [32-102](#)
- timers, [20-22](#)
 - watchdog, [20-31](#)
 - WIDTH_CAP mode, [31-23](#)
- time stamps, CAN, [27-21](#)
- timing
 - multichannel transfer, [30-18](#)
 - SPI, [28-8](#)
- timing examples, for SPORTs, [30-41](#)
- timing parameters, CAN, [27-12](#)
- TIMOD[1:0] field, [28-21](#), [28-26](#), [28-28](#), [28-45](#)
- TIMODx (transfer initiation mode) bits, [28-45](#)
- TKR (select DIOR negated pulsewidth) bits, [24-66](#)

- TKW (select DIOW negated pulsewidth) bits, [24-66](#)
- TLSBIT bit, [30-51](#), [30-53](#)
- TMLI (select interlock time) bits, [24-68](#)
- TM_PLL_VCO (boost PLL amplitude) bit, [32-132](#)
- TM_SEL (increase PLL charge pump current) bit, [32-132](#)
- TM_SHORT_CHAIN (shorten startup counter chain) bit, [32-132](#)
- TM (time from address valid to DIOR/DIOW) bits, [24-66](#)
- tools, development, [20-36](#)
- TOVF bit, [30-62](#), [30-66](#), [30-67](#)
- TPOLC bit, [31-50](#), [31-51](#)
- transfer count (ECCCNT) bits, [25-25](#)
- transfer direction (XFER_DIR) bit, [24-50](#)
- transfer initiate command, [28-27](#)
- transfer initiation from SPI master, [28-28](#)
- transfer initiation mode (TIMODx) bits, [28-45](#)
- transfer length (XFER_LENGTH) bits, [24-59](#)
- transfer size (TxferSize), [32-24](#), [32-27](#)
- transmission count reset (TFRCNT_RST) bit, [24-50](#)
- transmission error, SPI, [28-25](#)
- transmission error (TXE) bit, [28-48](#)
- transmit active mask (TX_ACT_MASK) bit, [23-33](#)
- transmit active (TX_ACT) bit, [23-30](#)
- transmit clock, serial (TSCLKx) pins, [30-31](#)
- transmit collision error, SPI, [28-25](#)
- transmit collision error (TXCOL) bit, [28-48](#)
- transmit data[15:0] field, [30-63](#)
- transmit data[31:16] field, [30-63](#)
- transmit data available mask (TX_DAT_RDY_MASK) bit, [23-33](#)
- transmit data available (TX_DAT_RDY) bit, [23-31](#)
- transmit data buffer[15:0] field, [28-48](#)
- transmit FIFO empty mask (TX_FIFO_ZERO_MASK) bit, [23-33](#)
- transmit FIFO empty (TX_FIFO_ZERO) bit, [23-31](#)
- transmit FIFO full mask (TX_FIFO_FULL_MASK) bit, [23-33](#)
- transmit FIFO full (TX_FIFO_FULL) bit, [23-31](#)
- transmit FIFO status mask (TX_FIFO_STAT_MASK) bit, [23-33](#)
- transmit FIFO status (TX_FIFO_STAT) bit, [23-30](#)
- Transmit FOT, [21-16](#)
- transmit hold[7:0] field, [31-41](#), [31-42](#)
- transmit underrun mask (TX_UNDERRUN_MASK) bit, [23-33](#)
- transmit underrun status (TX_UNDERRUN_STAT) bit, [23-32](#)
- transmit underrun (TX_UNDERRUN) bit, [23-30](#)
- TRFST bit, [30-52](#), [30-56](#)
- TRM bit, [27-47](#)
- TRRn bit, [27-79](#)
- TRSn bit, [27-78](#)
- TSCLKx signal, [30-5](#)
- TSEG1[3:0] field, [27-12](#), [27-49](#)
- TSEG2[2:0] field, [27-12](#), [27-49](#)
- TSFSE bit, [30-13](#), [30-52](#), [30-56](#)
- TSPEN bit, [30-11](#), [30-51](#), [30-52](#), [30-53](#)
- TSS (select time from STROBE edge to negation of DMARQ or assertion of STOP) bits, [24-68](#)

Index

- TSV[15:0] field, [27-61](#)
- tuning of DPHY clocks (CNOS) bits, [32-130](#)
- TUVF bit, [30-39](#), [30-62](#), [30-66](#), [30-67](#)
- TWI, [20-16](#), [29-1](#) to [29-67](#)
 - block diagram, [29-3](#)
 - bus arbitration, [29-8](#)
 - clock generation, [29-7](#)
 - electrical specifications, [29-67](#)
 - fast mode, [29-11](#)
 - features, [29-2](#)
 - general call address, [29-10](#)
 - general setup, [29-22](#)
 - I²C compatibility, [20-16](#)
 - master mode clock setup, [29-24](#)
 - master mode receive, [29-25](#)
 - master mode transmit, [29-24](#)
 - peripheral interface, [29-6](#)
 - pins, [29-5](#)
 - registers, list of, [29-34](#), [A-33](#)
 - slave mode operation, [29-22](#)
 - start and stop conditions, [29-9](#)
 - synchronization, [29-7](#)
 - transfer protocol, [29-7](#)
- TWI_CLKDIV (SCL clock divider) register, [29-36](#)
- TWI_CONTROL (TWI control) register, [29-36](#)
- TWI_ENA bit, [29-36](#)
- TWI_FIFO_CTL (TWI FIFO control) register, [29-45](#)
- TWI FIFO receive data double byte (TWI_RCV_DATA16 register), [29-54](#)
- TWI FIFO receive data single byte (TWI_RCV_DATA8 register), [29-53](#)
- TWI_FIFO_STAT (TWI FIFO status register), [29-17](#)
- TWI_FIFO_STAT (TWI FIFO status) register, [29-47](#)
- TWI FIFO status register (TWI_FIFO_STAT), [29-17](#)
- TWI FIFO transmit data double byte (TWI_XMT_DATA16) register, [29-52](#)
- TWI FIFO transmit data single byte (TWI_XMT_DATA8 register), [29-52](#)
- TWI interrupt mask (TWI_INT_MASK) register, [29-47](#)
- TWI interrupt status register (TWI_INT_STAT), [29-18](#)
- TWI_INT_MASK (TWI interrupt mask) register, [29-47](#)
- TWI_INT_STAT (TWI interrupt status register), [29-18](#)
- TWI_INT_STAT (TWI interrupt status) register, [29-51](#)
- TWI_MASTER_ADDR (TWI master mode address) register, [29-44](#)
- TWI master mode status register (TWI_MASTER_STAT), [29-13](#)
- TWI master mode status (TWI_MASTER_STAT) register, [29-45](#)
- TWI_MASTER_STAT (TWI master mode status register), [29-13](#)
- TWI_MASTER_STAT (TWI master mode status) register, [29-45](#)
- TWI_RCV_DATA16 (TWI FIFO receive data double byte) register, [29-54](#)
- TWI_RCV_DATA8 (TWI FIFO receive data single byte) register, [29-53](#)
- TWI_SLAVE_ADDR (TWI slave mode address) register, [29-39](#)
- TWI_SLAVE_CTL (TWI slave mode control) register, [29-37](#)
- TWI slave mode control (TWI_SLAVE_CTL) register, [29-37](#)
- TWI slave mode status register (TWI_SLAVE_STAT), [29-16](#)

- TWI slave mode status
(TWI_SLAVE_STAT) register,
[29-40](#)
- TWI_SLAVE_STAT (TWI slave mode
status register), [29-16](#)
- TWI_SLAVE_STAT (TWI slave mode
status) register, [29-40](#)
- TWI_XMT_DATA16 (TWI FIFO
transmit data double byte) register,
[29-52](#)
- TWI_XMT_DATA8 (TWI FIFO transmit
data single byte) register, [29-52](#)
- two-wire interface, *See* TWI
- TX_ACT_MASK (transmit active mask)
bit, [23-33](#)
- TX_ACT (transmit active) bit, [23-30](#)
- TXCOL bit, [28-48](#)
- TXCOL flag, [28-25](#)
- TXCOL (transmit collision error) bit,
[28-48](#)
- TX_COUNT (USB Tx byte count) bits,
[32-121](#)
- TX_DAT_RDY_MASK (transmit data
available mask) bit, [23-33](#)
- TX_DAT_RDY (transmit data available)
bit, [23-31](#)
- TXE bit, [28-25](#), [28-48](#)
- TXECNT[7:0] field, [27-89](#)
- TXE (transmission error) bit, [28-48](#)
- TXF bit, [30-62](#), [30-66](#), [30-67](#)
- TxferSize (transfer size), [32-24](#), [32-27](#)
- TX_FIFO_FULL_MASK (transmit FIFO
full mask) bit, [23-33](#)
- TX_FIFO_FULL (transmit FIFO full) bit,
[23-31](#)
- TX_FIFO_STAT_MASK (transmit FIFO
status mask) bit, [23-33](#)
- TX_FIFO_STAT (transmit FIFO status)
bit, [23-30](#)
- TX_FIFO_ZERO_MASK (transmit FIFO
empty mask) bit, [23-33](#)
- TX_FIFO_ZERO (transmit FIFO empty)
bit, [23-31](#)
- TX hold register, [30-62](#)
- TXHRE bit, [30-67](#)
- TX_INTx_R (route Tx IRQ to INTx) bits,
[32-94](#)
- TxPktRdy autoselect enable (AUTOSET_T)
bit, [32-102](#)
- TXPKTRDY (data packet in FIFO
indicator) bit, [32-98](#)
- TXPKTRDY_T (data packet in FIFO
indicator) bit, [32-102](#)
- TX_POLL_INTERVAL (USB Tx poll
interval) bits, [32-118](#)
- Tx protocol type (PROTOCOL_T) bits,
[32-117](#)
- TXREQ signal, [31-8](#)
- TXS bit, [28-31](#), [28-48](#)
- TXSE bit, [30-52](#), [30-55](#)
- TXS (SPIx_TDBR data buffer status) bit,
[28-48](#)
- TX_UNDERRUN_MASK (transmit
underrun mask) bit, [23-33](#)
- TX_UNDERRUN_STAT (transmit
underrun status) bit, [23-32](#)
- TX_UNDERRUN (transmit underrun)
bit, [23-30](#)
- TZAH (select minimum delay required for
output) bits, [24-69](#)

Index

U

- UART, [31-1](#) to [31-60](#)
 - autobaud detection, [31-21](#), [31-54](#)
 - baud rate, [31-9](#)
 - baud rate examples, [31-20](#)
 - bit rate examples, [31-20](#)
 - bitstream, [31-7](#)
 - block diagram, [31-3](#), [31-12](#)
 - booting, [31-21](#)
 - character transmission, [31-55](#)
 - clock, [31-19](#)
 - code examples, [31-52](#)
 - data words, [31-6](#)
 - divisor reset, [31-49](#)
 - DMA channels, [31-25](#)
 - DMA mode, [31-25](#)
 - errors during reception, [31-10](#)
 - external interfaces, [31-4](#)
 - features, [31-2](#)
 - glitch filtering, [31-15](#)
 - initialization, [31-52](#)
 - internal interfaces, [31-5](#)
 - interrupt channels, [31-43](#)
 - interrupt conditions, [31-46](#)
 - interrupts, [31-17](#)
 - IrDA mode, [31-2](#)
 - IrDA receiver, [31-15](#)
 - IrDA receiver pulse detection, [31-16](#)
 - IrDA transmit pulse, [31-15](#)
 - IrDA transmitter, [31-14](#)
 - and ISRs, [31-24](#)
 - loopback mode, [31-33](#)
 - mixing modes, [31-27](#)
 - non-DMA interrupt operation, [31-57](#)
 - non-DMA mode, [31-23](#)
 - receive operation, [31-9](#)
 - receive sampling window, [31-15](#)
 - registers, table, [31-28](#)
 - signals, [31-4](#)
 - standard, [31-1](#)
 - string transmission, [31-56](#)
 - switching from DMA to non-DMA, [31-27](#)
 - switching from non-DMA to DMA, [31-27](#)
 - and system DMA, [31-44](#)
 - transmission, [31-8](#)
 - transmission SYNC bit use, [31-58](#)
- UART divisor latch high byte
 - (UARTx_DLH) registers, [31-48](#)
- UART divisor latch low byte
 - (UARTx_DLL) registers, [31-48](#)
- UART global control (UARTx_GCTL) registers, [31-50](#)
- UART interrupt enable clear
 - (UARTx_IER_CLEAR) registers, [31-44](#)
- UART interrupt enable registers
 - (UARTx_IER), [31-45](#)
- UART interrupt enable set
 - (UARTx_IER_SET) registers, [31-44](#)
- UART interrupt enable (UARTx_IER) registers, [31-43](#)
- UART line control registers
 - (UARTx_LCR), [31-30](#)
- UART line control (UARTx_LCR) registers, [31-30](#)
- UART line status registers (UARTx_LSR), [31-36](#)
- UART line status (UARTx_LSR) registers, [31-36](#)
- UART modem control (UARTx_MCR) registers, [31-33](#)
- UART modem status (UARTx_MSR) registers, [31-39](#)
- UART receive buffer registers
 - (UARTx_RBR), [31-9](#)
- UART receive buffer (UARTx_RBR) registers, [31-42](#)

- UART scratch registers (UARTx_SCR),
31-49
- UART scratch (UARTx_SCR) registers,
31-49
- UART transmit holding (UARTx_THR)
registers, 31-41
- UARTx, 31-28
- UARTx_DLH (UART divisor latch high
byte registers), 31-28
- UARTx_DLH (UART divisor latch high
byte) registers, 31-48
- UARTx_DLL (UART divisor latch low
byte registers), 31-28
- UARTx_DLL (UART divisor latch low
byte) registers, 31-48
- UARTx_GCTL (UART global control
registers), 31-29
- UARTx_GCTL (UART global control)
registers, 31-50
- UARTx_IER_CLEAR (UART interrupt
enable clear) registers, 31-44
- UARTx_IER_SET (UART interrupt
enable set) registers, 31-44
- UARTx_IER (UART interrupt enable
registers), 31-45
- UARTx_IER (UART interrupt enable)
registers, 31-43
- UARTx_IIR (UART interrupt
identification registers), 31-29
- UARTx_LCR (UART line control
registers), 31-29, 31-30
- UARTx_LCR (UART line control)
registers, 31-30
- UARTx_LSR (UART line status registers),
31-29, 31-36
- UARTx_LSR (UART line status) registers,
31-36
- UARTx_MCR (UART modem control
registers), 31-29
- UARTx_MCR (UART modem control)
registers, 31-33
- UARTx_MSR (UART modem status)
registers, 31-39
- UARTx_RBR (UART receive buffer
registers), 31-9, 31-29
- UARTx_RBR (UART receive buffer)
registers, 31-42
- UARTx_SCR (UART scratch registers),
31-29, 31-49
- UARTx_SCR (UART scratch) registers,
31-49
- UARTx_THR (UART transmit holding
registers), 31-8, 31-29
- UARTx_THR (UART transmit holding)
registers, 31-41
- UCCNF[3:0] field, 27-29, 27-87
- UCCNT[15:0] field, 27-88
- UCCT bit, 27-87
- UCE bit, 27-87
- UCEIF bit, 27-27, 27-52
- UCEIM bit, 27-27, 27-50
- UCEIS bit, 27-27, 27-51
- UCEN bit, 31-8, 31-19, 31-50, 31-51
- UCRC[15:0] field, 27-88
- UCRC bit, 27-87
- UDMAIN_CSTATE (ultra DMA-In
mode state machine current state) bits,
24-61
- UDMAIN_DONE_INT (ultra-DMA in
transfer done interrupt status) bit,
24-58
- UDMAIN_DONE_MASK (ultra-DMA
in transfer done interrupt mask) bit,
24-56
- UDMAIN_FIFO_THRS (ultra DMA-IN
FIFO threshold) bits, 24-50

Index

- UDMA_IN_FL (ultra DMA input FIFO level) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-68](#), [24-69](#)
- UDMAIN_TERM_INT (device terminate ultra-DMA-in transfer interrupt status) bit, [24-58](#)
- UDMAIN_TERM_MASK (device terminate ultra-DMA-in transfer interrupt mask) bit, [24-56](#)
- UDMAIN_TFRCNT (UDMA in transfer count) bits, [24-63](#)
- UDMAOUT_CSTATE (ATAPI IORDY line status) bits, [24-61](#)
- UDMAOUT_DONE_INT (ultra-DMA out transfer done interrupt status) bit, [24-58](#)
- UDMAOUT_DONE_MASK (ultra-DMA out transfer done interrupt mask) bit, [24-56](#)
- UDMAOUT_TERM_INT (device terminate ultra-DMA-out transfer interrupt status) bit, [24-58](#)
- UDMAOUT_TERM_MASK (device terminate ultra-DMA-out transfer interrupt mask) bit, [24-56](#)
- UDMAOUT_TFRCNT (UDMA out transfer count) bits, [24-64](#)
- UDMA_XFER_ON (ultra DMA transfer in progress) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- UIAIF bit, [27-27](#), [27-52](#)
- UIAIM bit, [27-27](#), [27-50](#)
- UIAIS bit, [27-27](#), [27-51](#)
- ultra DMA-IN FIFO threshold (UDMAIN_FIFO_THRS) bits, [24-50](#)
- ultra DMA-In mode state machine current state (UDMAIN_CSTATE) bits, [24-61](#)
- ultra DMA input FIFO level (UDMA_IN_FL) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-68](#), [24-69](#)
- ultra DMA input FIFO level (ULTRA_IN_FL) bits, [24-52](#)
- ultra-DMA in transfer done interrupt mask (UDMAIN_DONE_MASK) bit, [24-56](#)
- ultra-DMA in transfer done interrupt status (UDMAIN_DONE_INT) bit, [24-58](#)
- ultra-DMA out transfer done interrupt mask (UDMAOUT_DONE_MASK) bit, [24-56](#)
- ultra-DMA out transfer done interrupt status (UDMAOUT_DONE_INT) bit, [24-58](#)
- ultra DMA transfer in progress (UDMA_XFER_ON) bit, [24-52](#), [24-53](#), [24-54](#), [24-55](#), [24-59](#), [24-60](#), [24-61](#), [24-62](#), [24-63](#), [24-64](#), [24-65](#), [24-66](#), [24-67](#), [24-68](#), [24-69](#)
- ultra DMA transfer in progress (ULTRA_XFER_ON) bit, [24-52](#)
- ULTRA_IN_FL (ultra DMA input FIFO level) bits, [24-52](#)
- ULTRA_START (start ultra-DMA Op) bit, [24-50](#)
- ULTRA_XFER_ON (ultra DMA transfer in progress) bit, [24-52](#)
- UNDERRUN_T (no TxPktRdy for IN token) bit, [32-102](#)
- unframed/framed, serial data, [30-32](#)
- universal asynchronous receiver/transmitter, *See* UART

- universal counter, CAN, [27-29](#)
- universal counter exceeded interrupt, CAN, [27-27](#)
- USB_APHY_CNTRL2 (USB APHY control 2) register, [32-130](#)
- USB APHY control 2 (USB_APHY_CNTRL2) register, [32-130](#)
- USB common interrupts enable (USB_INTRUSBE) register, [32-90](#)
- USB common interrupts (USB_INTRUSB) register, [32-89](#)
- USB control/status EP0 (USB_CSR0) register, [32-98](#)
- USB_COUNT0 (USB received byte count in EP0 FIFO) register, [32-115](#)
- USB_CSR0 (USB control/status EP0) register, [32-98](#)
- USB DMA endpoint x interrupt (DMAx_INT) bits, [32-134](#)
- USB_DMA_INTERRUPT (USB DMA interrupt) register, [32-134](#)
- USB DMA interrupt (USB_DMA_INTERRUPT) register, [32-134](#)
- USB DMAx address high (USB_DMAxADDRHIGH) register, [32-138](#)
- USB DMAx address low (USB_DMAxADDRLOW) register, [32-137](#)
- USB_DMAxADDRHIGH (USB DMAx address high) register, [32-138](#)
- USB_DMAxADDRLOW (USB DMAx address low) register, [32-137](#)
- USB_DMAxCONTROL (USB DMAx control) registers, [32-135](#)
- USB DMAx control (USB_DMAxCONTROL) registers, [32-135](#)
- USB_DMAxCOUNTHIGH (USB DMAx count high) register, [32-140](#)
- USB DMAx count high (USB_DMAxCOUNTHIGH) register, [32-140](#)
- USB_DMAxCOUNTLOW (USB DMAx count low) register, [32-139](#)
- USB DMAx count low (USB_DMAxCOUNTLOW) register, [32-139](#)
- USB enable (GLOBAL_ENA) bit, [32-95](#)
- USB endpoint index (SELECTED_ENDPOINT) bits, [32-91](#), [32-93](#)
- USB_FADDR (USB function address) register, [32-81](#)
- USB frame number (FRAME_NUMBER) bits, [32-91](#)
- USB frame number (USB_FRAME) register, [32-91](#)
- USB_FRAME (USB frame number) register, [32-91](#)
- USB_FS_EOF1 (USB full-speed EOF 1) register, [32-128](#)
- USB full-speed EOF 1 (USB_FS_EOF1) register, [32-128](#)
- USB function address (USB_FADDR) register, [32-81](#)
- USB global control (USB_GLOBAL_CTL) register, [32-95](#)
- USB_GLOBAL_CTL (USB global control) register, [32-95](#)
- USB global interrupt (USB_GLOBINTR) register, [32-94](#)
- USB_GLOBINTR (USB global interrupt) register, [32-94](#)
- USB hibernate signal (CSR_HBR) bit, [32-130](#)

Index

- USB high-speed EOF 1 (USB_HS_EOF1) register, [32-128](#)
- USB_HS_EOF1 (USB high-speed EOF 1) register, [32-128](#)
- USB_INDEX (USB index) register, [32-91](#), [32-93](#)
- USB index (USB_INDEX) register, [32-91](#), [32-93](#)
- USB_INTRRXE (USB receive interrupt enable) register, [32-88](#)
- USB_INTRRX (USB receive interrupt) register, [32-86](#)
- USB_INTRTXE (USB transmit interrupt enable) register, [32-87](#)
- USB_INTRTX (USB transmit interrupt) register, [32-85](#)
- USB_INTRUSBE (USB common interrupts enable) register, [32-90](#)
- USB_INTRUSB (USB common interrupts) register, [32-89](#)
- USB_INTx_R (route USB/VBUS IRQ to INTx) bits, [32-94](#)
- USB_LINKINFO (USB link info) register, [32-127](#)
- USB link info (USB_LINKINFO) register, [32-127](#)
- USB low-speed EOF 1 (USB_LS_EOF1) register, [32-129](#)
- USB_LS_EOF1 (USB low-speed EOF 1) register, [32-129](#)
- USB max Rx data in frame (MAX_PACKET_SIZE_R) bits, [32-107](#)
- USB max Tx data in frame (MAX_PACKET_SIZE_T) bits, [32-97](#)
- USB_NAKLIMIT0 (USB NAK limit 0) register, [32-117](#)
- USB NAK limit 0 (USB_NAKLIMIT0) register, [32-117](#)
- USB or non-USB part (USBPARTB1V) bit, [32-130](#)
- USB OTG
 - DMA master channels, [32-57](#)
 - features, [32-2](#)
 - host negotiation /configuration, [32-49](#)
 - interface pins, [32-45](#)
 - OTG session request, [32-47](#)
 - peripheral mode operation, [32-13](#)
 - transferring packets using DMA, [32-59](#)
- USB_OTG_DEV_CTL (USB OTG device control) register, [32-122](#), [32-124](#)
- USB OTG device control (USB_OTG_DEV_CTL) register, [32-122](#), [32-124](#)
- USB_OTG_VBUS_MASK (USB OTG VBUS mask) register, [32-126](#)
- USB OTG VBUS mask (USB_OTG_VBUS_MASK) register, [32-126](#)
- USBPARTB1V (USB part or non-USB part) bit, [32-130](#)
- USB peripheral device address (FUNCTION_ADDRESS) bits, [32-81](#)
- USB PLL OSC control (USB_PLLOSC_CTRL) register, [32-132](#)
- USB_PLLOSC_CTRL (USB PLL OSC control) register, [32-132](#)
- USB power management (USB_POWER) register, [32-82](#)
- USB_POWER (USB power management) register, [32-82](#)
- USB pu/pd restore control (CSR_RSTD) bit, [32-130](#)
- USB received byte count in EP0 FIFO (USB_COUNT0) register, [32-115](#)

Index

- USB receive interrupt enable
(USB_INTRRXE) register, [32-88](#)
- USB receive interrupt (USB_INTRRX)
register, [32-86](#)
- USB reset (RESET) bit, [32-82](#)
- USB Rx byte count (RX_COUNT) bits,
[32-116](#)
- USB Rx byte count (USB_RXCOUNT)
register, [32-116](#)
- USB Rx control/status EPx (USB_RXCSR)
register, [32-109](#)
- USB_RXCOUNT (USB Rx byte count)
register, [32-116](#)
- USB_RXCSR (USB Rx control/status EPx)
register, [32-109](#)
- USB Rx endpoint x interrupt enable
(EPx_RX_E) bits, [32-88](#)
- USB Rx endpoint x interrupt (EPx_RX)
bits, [32-86](#)
- USB_RXINTERVAL (USB Rx interval)
register, [32-120](#)
- USB Rx interval (USB_RXINTERVAL)
register, [32-120](#)
- USB_RX_MAX_PACKET (USB Rx max
packet) register, [32-107](#)
- USB Rx max packet
(USB_RX_MAX_PACKET) register,
[32-107](#)
- USB Rx poll interval
(RX_POLL_INTERVAL) bits,
[32-120](#)
- USB_RXTYPE (USB Rx type) register,
[32-119](#)
- USB Rx type (USB_RXTYPE) register,
[32-119](#)
- USB_SRP_CLKDIV (USB SRP clock
divider) register, [32-133](#)
- USB SRP clock divider
(USB_SRP_CLKDIV) register,
[32-133](#)
- USB transmit interrupt enable
(USB_INTRTXE) register, [32-87](#)
- USB transmit interrupt (USB_INTRTX)
register, [32-85](#)
- USB Tx byte count (TX_COUNT) bits,
[32-121](#)
- USB Tx byte count (USB_TXCOUNT)
register, [32-121](#)
- USB Tx control/status EPx (USB_TXCSR)
register, [32-102](#)
- USB_TXCOUNT (USB Tx byte count)
register, [32-121](#)
- USB_TXCSR (USB Tx control/status EPx)
register, [32-102](#)
- USB Tx endpoint x interrupt enable
(EPx_TX_E) bits, [32-87](#)
- USB Tx endpoint x interrupt (EPx_TX)
bits, [32-85](#)
- USB_TXINTERVAL (USB Tx interval)
register, [32-118](#)
- USB Tx interval (USB_TXINTERVAL)
register, [32-118](#)
- USB_TX_MAX_PACKET (USB Tx max
packet) register, [32-97](#)
- USB Tx max packet
(USB_TX_MAX_PACKET) register,
[32-97](#)
- USB Tx poll interval
(TX_POLL_INTERVAL) bits,
[32-118](#)
- USB_TXTYPE (USB Tx type) register,
[32-117](#)
- USB Tx type (USB_TXTYPE) register,
[32-117](#)
- USB VBUS pulse length (USB_VPLEN)
register, [32-127](#)
- USB_VPLEN (USB VBUS pulse length)
register, [32-127](#)

Index

V

VBUS1–0 (VBUS level indicator) bit, [32-122](#), [32-124](#)
VBUS_ERROR_BE (VBus threshold IRQ enable) bit, [32-90](#)
VBUS_ERROR_B (VBus threshold indicator) bit, [32-89](#)
VBUS level indicator (VBUS1–0) bit, [32-122](#), [32-124](#)
VBUS pulse length (VPLEN) bits, [32-127](#)
VBus threshold indicator (VBUS_ERROR_B) bit, [32-89](#)
VBus threshold IRQ enable (VBUS_ERROR_BE) bit, [32-90](#)
VDK, [20-38](#)
VisualDSP++, [20-36](#)
 debugger, [20-37](#)
voltage regulator, [20-34](#)
VPLEN (VBUS pulse length) bits, [32-127](#)
VR_CTL (voltage regulator control register), [27-41](#)

W

wait for connect (WTCON) bits, [32-127](#)
wait from IDPULLUP (WTID) bits, [32-127](#)
wake-up interrupt, CAN, [27-27](#)
Wake-up Preamble Detected interrupt enable, [21-45](#)
Wake-Up Preamble Received (WUP) interrupt event, [21-34](#)
Wake-Up (WAKEUP) bit, [21-17](#)
watchdog mode, CAN, [27-21](#)
watchdog timer, [20-31](#)
WBA bit, [27-46](#)
WB_EDGE (write buffer edge detect) bit, [25-21](#), [25-22](#)
WB_FULL (write buffer full) bit, [25-21](#)

WB_OVF (write buffer overflow) bit, [25-22](#)
wide bus mode enable (WIDE_BUS) bit, [23-23](#)
WIDE_BUS (wide bus mode enable) bit, [23-23](#)
WLS[1:0] field, [31-30](#)
WOFF[9:0] field, [30-24](#), [30-70](#)
WOM bit, [28-20](#), [28-45](#)
WOM (write open drain master) bit, [28-45](#)
word length
 SPI, [28-21](#)
 SPORT, [30-29](#)
 SPORT receive data, [30-64](#)
 SPORT transmit data, [30-62](#)
WR bit, [27-47](#)
WR_DLY (write strobe delay) bits, [25-20](#)
WR_DONE (page write done) bit, [25-22](#)
write buffer edge detect (WB_EDGE) bit, [25-21](#), [25-22](#)
write buffer full (WB_FULL) bit, [25-21](#)
write buffer overflow (WB_OVF) bit, [25-22](#)
write open drain master (WOM) bit, [28-45](#)
write strobe delay (WR_DLY) bits, [25-20](#)
WSIZE[3:0] field, [30-23](#), [30-70](#)
WT bit, [27-47](#)
WTCON (wait for connect) bits, [32-127](#)
WTID (wait from IDPULLUP) bits, [32-127](#)
WUIF bit, [27-27](#), [27-52](#)
WUIM bit, [27-27](#), [27-50](#)
WUIS bit, [27-27](#), [27-51](#)
WUPEN, [21-45](#)

X

XFER_DIR (transfer direction) bit, [24-50](#)
XFER_LENGTH (transfer length) bits, [24-59](#)
XFR_TYPE (operating mode) bits, [26-82](#)

XMTDATA16[15:0] field, [29-53](#)
XMTDATA8[7:0] field, [29-52](#)
XMTFLUSH bit, [29-45](#), [29-47](#)
XMTINTLEN bit, [29-45](#), [29-46](#)
XMTSERV bit, [29-19](#), [29-51](#)
XMTSERVM bit, [29-48](#), [29-49](#)
XMTSTAT[1:0] field, [29-17](#), [29-47](#)

XOFF (transmitter off) bit, [31-33](#)

Y

YFIFO_ERR (luma FIFO error) bit, [26-89](#)
YFIFO_ERR (Luma FIFO Overflow
Error) bit, [24-50](#), [24-56](#), [24-57](#)

