ADSP-BF539 Blackfin Processor Hardware Reference

Revision 1.1, February 2013

Part Number
82-000003-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

OPERATING MODES AND STATES

PROGRAM SEQUENCER

CHIP BUS HIERARCHY

DYNAMIC POWER MANAGEMENT

DIRECT MEMORY ACCESS

SPI COMPATIBLE PORT CONTROLLERS

PARALLEL PERIPHERAL INTERFACE

SERIAL PORT CONTROLLERS

PROGRAMMABLE FLAGS

EXTERNAL BUS INTERFACE UNIT

CAN MODULE

MEDIA TRANSCEIVER MODULE (MXVR)

SYSTEM DESIGN

NUMERIC FORMATS

PREFACE

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

Purpose of This Manual

ADSP-BF539 Blackfin Processor Hardware Reference contains information about the architecture for the ADSP-BF539 processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.

For programming information, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see *ADSP-BF539/ADSP-BF539F Embedded Processor Data Sheet*.

Intended Audience

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

Manual Contents

This manual contains:

- Chapter 1, "Introduction"
  Provides a high level overview of the processor. Architectural descriptions include functional blocks, buses, and ports, including features and processes they support.
- Chapter 2, "Computational Units"
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.
- Chapter 3, "Operating Modes and States"
  Describes the three operating modes of the processor: Emulation mode, Supervisor mode, and User mode. The chapter also describes Idle state and Reset state.
- Chapter 4, "Program Sequencer"
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.
- Chapter 5, "Data Address Generators"
  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and Pointer registers, memory address alignment, and DAG instructions.
- Chapter 6, "Memory"
  Describes L1 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). Discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core.

- Chapter 7, "Chip Bus Hierarchy"
  Describes on-chip buses, including how data moves through the system. The chapter also discusses the system memory map, major system components, and the system interconnects.
- Chapter 8, "Dynamic Power Management"
  Describes system reset and power-up configuration, system clocking and control, and power management.
- Chapter 9, "Direct Memory Access"
  Describes the peripheral DMA and memory DMA controllers. The peripheral DMA section discusses direct, block data movements between a peripheral with DMA access and internal or external memory spaces.

  The memory DMA section discusses memory-to-memory transfer capabilities among the processor memory spaces and the L1, external synchronous, and asynchronous memories.
- Chapter 10, "SPI Compatible Port Controllers"
  Describes the serial peripheral interface (SPI) ports that provide an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 11, "Parallel Peripheral Interface"
  Describes the parallel peripheral interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and used for digital video and data converter applications.
- Chapter 12, "Serial Port Controllers"
  Describes the independent, synchronous serial port controllers that provide an I/O interface to a variety of serial peripheral devices.
- Chapter 13, "UART Port Controllers"
  Describes the Universal Asynchronous Receiver/Transmitter (UART) ports, which convert data between serial and parallel for-

mats and includes modem control and interrupt handling hardware. The UARTs support the half-duplex IrDA® SIR protocol as a mode-enabled feature.

- Chapter 14, "Programmable Flags"
  Describes the programmable flags, including how to configure the pins as inputs and outputs and how to generate interrupts.
- Chapter 15, "General Purpose Input/Output Ports"
  Describes the general purpose I/O pins, including how to configure the pins as inputs and outputs.
- Chapter 16, "Timers"
  Describes the general purpose timers that can be configured in any of three modes; the core timer that can generate periodic interrupts for a variety of timing functions; and the watchdog timer that can implement software watchdog functions, such as generating events to the Blackfin processor core.
- Chapter 17, "Real-Time Clock"
  Describes a set of digital watch features of the processor, including time of day, alarm, and stopwatch countdown.
- Chapter 18, "External Bus Interface Unit"
  Describes the External Bus Interface Unit of the processor. The chapter also discusses the asynchronous memory interface, the SDRAM controller (SDC), related registers, and SDC configuration and commands.
- Chapter 19, "CAN Module"
  Describes the CAN module, a low bit rate serial interface intended for use in applications where bit rates are typically up to 1Mbit/s.
- Chapter 20, "Two-Wire Interface Controllers"
  Describes the Two-Wire Interface (TWI) controller, which allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification* version 2.1 dated January 2000.

- Chapter 21, "Media Transceiver Module (MXVR)"
  Describes the Media Transceiver Module of the processor, which 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.
- Chapter 22, "System Design"
  Describes how to use the processor as part of an overall system. It includes information about interfacing the processor to external memory chips, bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Chapter 23, "Blackfin Processor Debug"
  Describes the Blackfin processor debug functionality, which can be used for software debugging and complements some services often found in an operating system.
- Appendix A, "Blackfin Processor Core MMR Assignments"
  Lists the core memory-mapped registers, their addresses, and cross-references to text.
- Appendix B, "System MMR Assignments"
  Lists the system memory-mapped registers, their addresses, and cross-references to text.
- Appendix C, "Test Features"
  Describes test features for the processor; discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.
- Appendix D, "Numeric Formats"
  Describes various aspects of the 16-bit data format. The chapter also describes how to implement a block floating-point format in software.

What's New in This Manual

This is Revision 1.1 of the *ADSP-BF539 Blackfin Processor Hardware Reference*. This revision corrects minor typographical errors and the following issues:

- UART not half-duplex in Chapter 1, "Introduction"
- Core Double Fault Reset Enable bit (DOUBLE_FAULT) set in the SWRST register in Chapter 3, "Operating Modes and States"
- RETI instructions need not be first in nested interrupts and complete table of hardware conditions causing hardware interrupts in Chapter 4, "Program Sequencer"
- Core priority over DMA when accessing L1 SRAM in Chapter 7, "Chip Bus Hierarchy"
- Removal of reference to datasheet, note on programming the STOPCK bit, and description of the WAKE bit in Chapter 8, "Dynamic Power Management"
- Obsolete DMA error address range deleted in Chapter 9, "Direct Memory Access"
- Termination of SPI TX DMA operations in Chapter 10, "SPI Compatible Port Controllers"
- Behavior on startup when using an external clock and receiver and transmitter enable bit names standardized on RSPEN and TSPEN in Chapter 12, "Serial Port Controllers"
- Note on the TINT bit in the TCNTL register in Chapter 16, "Timers"
- Sampling the ARDY pin when it is asserted and note on timing dependencies for the TRP and TRAS settings in the EBIU_SDGCTL register in Chapter 18, "External Bus Interface Unit"

- Detection of recessive-to-dominant edges and note on `CAN_GIS` and `CAN_GIF` programming in Chapter 19, "CAN Module"
- Coverage of previously undocumented clock stretching behavior and miscellaneous changes across Chapter 20, "Two-Wire Interface Controllers"
- Clarification of watchpoint ranges in Chapter 23, "Blackfin Processor Debug"

Technical Support

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

- Post your questions in the processors and DSP support community at EngineerZone®:
  `http://ez.analog.com/community/dsp`
- Submit your questions to technical support directly at:
  `http://www.analog.com/support`
- E-mail your questions about processors, DSPs, and tools development software from **CrossCore® Embedded Studio** or **VisualDSP++®**:

  Choose **Help > Email Support**. This creates an e-mail to `processor.tools.support@analog.com` and automatically attaches your **CrossCore Embedded Studio** or **VisualDSP++** version information and `license.dat` file.
- E-mail your questions about processors and processor applications to:
  `processor.support@analog.com` or
  `processor.china@analog.com` (Greater China support)

- In the **USA only**, call **1-800-ANALOGD** (1-800-262-5643)
- Contact your Analog Devices sales office or authorized distributor. Locate one at:
  www.analog.com/adi-sales
- Send questions by mail to:
  Processors and DSP Technical Support
  Analog Devices, Inc.
  Three Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Supported Processors

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

Product Information

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

Analog Devices Web Site

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

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

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

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

EngineerZone

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

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

Notation Conventions

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the IDE environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

Register Diagram Conventions

Register diagrams use the following conventions:

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

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

The following figure shows an example of these conventions.

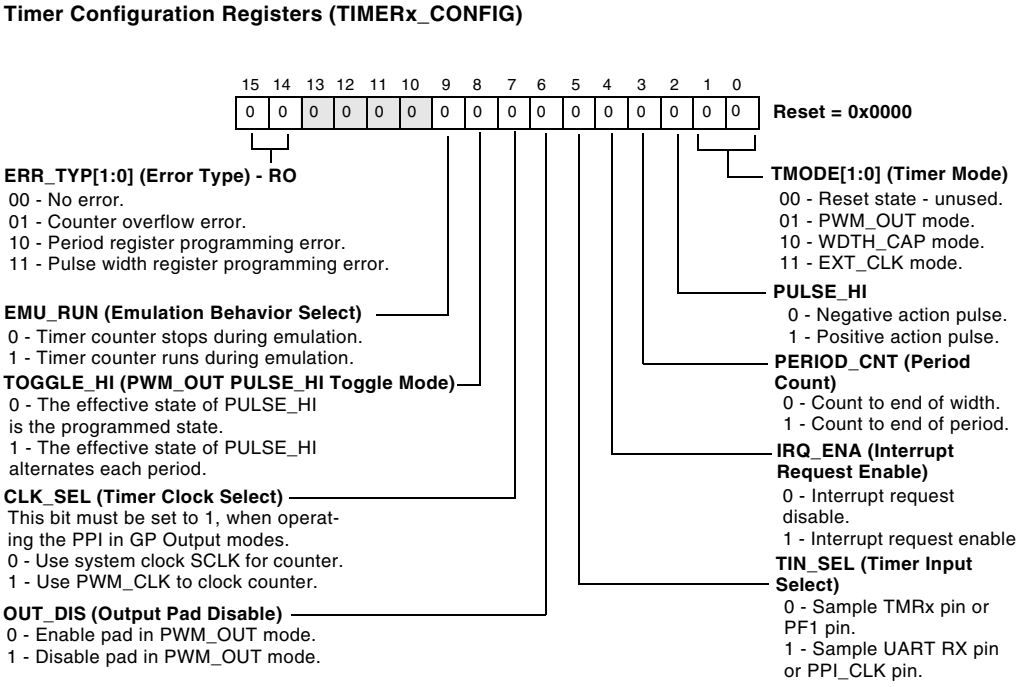


Figure 1. Register Diagram Example

1 INTRODUCTION

The ADSP-BF539 Blackfin® processor is derived from the ADSP-BF533 processor, offering similar performance and ease of use capabilities, but with enhanced peripheral features, targeted for the automotive and industrial markets. Common peripherals share the same features and functions.

Any time a processor is referenced by name (for example, ADSP-BF539), the information provided applies to the processor derivatives with on-chip flash memory as well (for example, ADSP-BF539F4 and ADSP-BF539F8).

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

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

Purpose of this Manual

This Blackfin processor hardware reference provides architectural information about enhanced Blackfin processors that include the ADSP-BF539 processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.

For programming information, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

Table 1-1 can be used to identify chapters from *ADSP-BF533 Blackfin Processor Hardware Reference* that are applicable to ADSP-BF539 Blackfin products.

For programmers familiar with the ADSP-BF533/ADSP-BF532/ ADSP-BF531 processors, the ADSP-BF539 will be very similar, as they are built from the same processor core. The ADSP-BF539 uses many of the same peripherals that are found on the ADSP-BF533/ADSP-BF532/ ADSP-BF531 processors (see Table 1-1 on page 1-3).

Table 1-1 on page 1-3 is intended as a guide that can be used to identify how chapters of this manual compare to the *ADSP-BF533 Blackfin Processor Hardware Reference* chapters, such that an experienced programmer does not need to read every chapter of this manual to understand the operation of the ADSP-BF539.

- No change—the reader can refer directly to the *ADSP-BF533 Blackfin Processor Hardware Reference* for this chapter.
- Changed—the *ADSP-BF533 Blackfin Processor Hardware Reference* chapter has been copied into this book, but some changes have been made or features added.
- New—this is an entirely new chapter and is the only source of reference for the material.

Table 1-1. Guide to Hardware Reference Chapter Differences

ADSP-BF539		ADSP-BF533		Comments
Chapter Number	Chapter Title	Chapter Number	Status	
1	Introduction	1	Changed	Describes added peripherals.
2	Computational Units	2	No changes	
3	Operating Modes and States	3	Changed	BMODE changes.
4	Program Sequencer	4	Changed	Describes additional interrupt sources.
5	Data Address Generators	5	No changes	
6	Memory	6	No changes	
7	Chip Bus Hierarchy	7	Changed	Changes block diagram and descriptions.
8	Dynamic Power Management	8	Small changes	Provides wake up from CAN and MXVR operation.
9	Direct Memory Access	9	Changed	Adds a second DMA controller.
10	SPI Compatible Port Controllers	10	Changed	Adds more SPI ports. Changes SPI slave select and slave enable functionality.
11	Parallel Peripheral Interface	11	No changes	
12	Serial Port Controllers	12	Changed	Adds more Serial ports.
13	UART Port Controller	13	Changed	Adds more UART ports.
14	Programmable Flags	14	No changes	
15	General Purpose I/O		New	Describes General Purpose I/O pins.
16	Timers	15	No changes	
17	Real Time Clock	16	No changes	
18	External Bus Interface Unit	17	No changes	

Table 1-1. Guide to Hardware Reference Chapter Differences (Cont'd)

ADSP-BF539		ADSP-BF533		Comments
Chapter Number	Chapter Title	Chapter Number	Status	
19	CAN Module		New	Describes the CAN 2.0B controller.
20	Two-Wire Interface Controllers		New	Describes the TWI controller connection to an I^2C© network.
21	Media Transceiver Module (MXVR)		New	Describes the MXVR interface to a MOST® network controller.
22	System Design	18	Changed	
23	Blackfin Processor Debug	19	No changes	

Peripherals

The processor system peripherals include the following:

- Parallel peripheral interface (PPI)
- Serial ports (SPORTs)
- Serial peripheral interfaces (SPI)
- Media transceiver (MXVR)—for connection to a MOST network
- Controller area network (CAN)
- Two-wire interfaces (TWI)—for connection to an I^2C network
- General-purpose timers
- Universal Asynchronous Receiver Transmitters (UART)
- Real-time clock (RTC)

- Watchdog timer
- General-purpose I/O (including programmable flags)

These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1 on page 1-5.

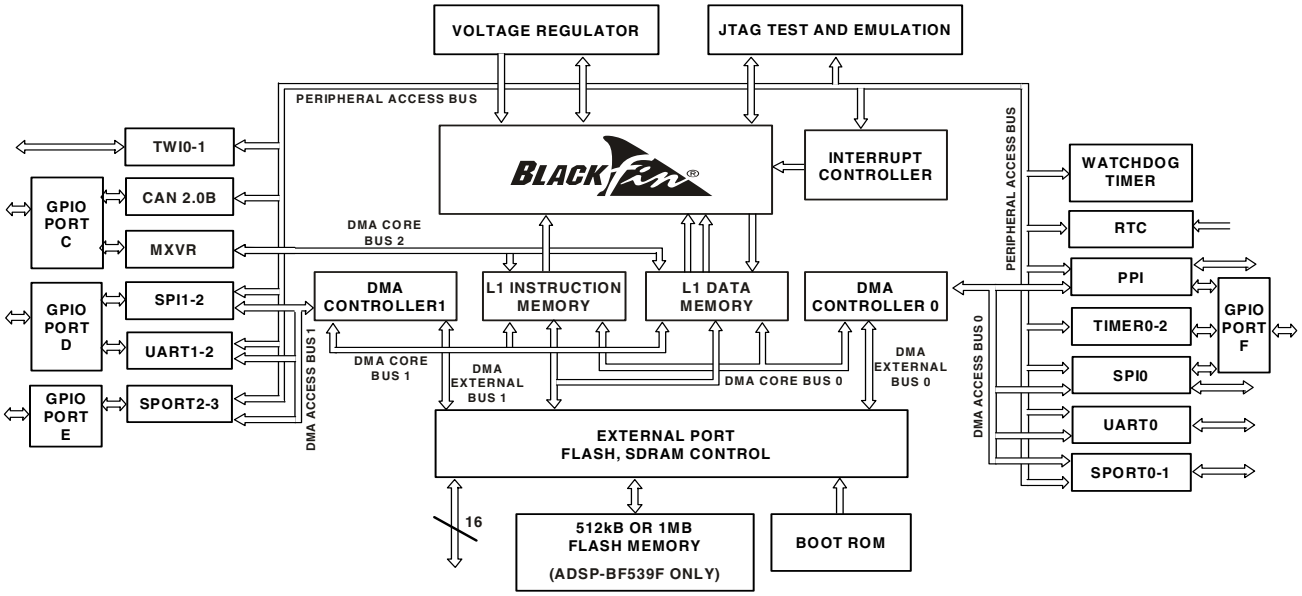


Figure 1-1. Processor Block Diagram

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

Core Architecture

The processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, as shown in Figure 1-2 on page 1-6. The computational units process 8-, 16-, or 32-bit data from the register file.

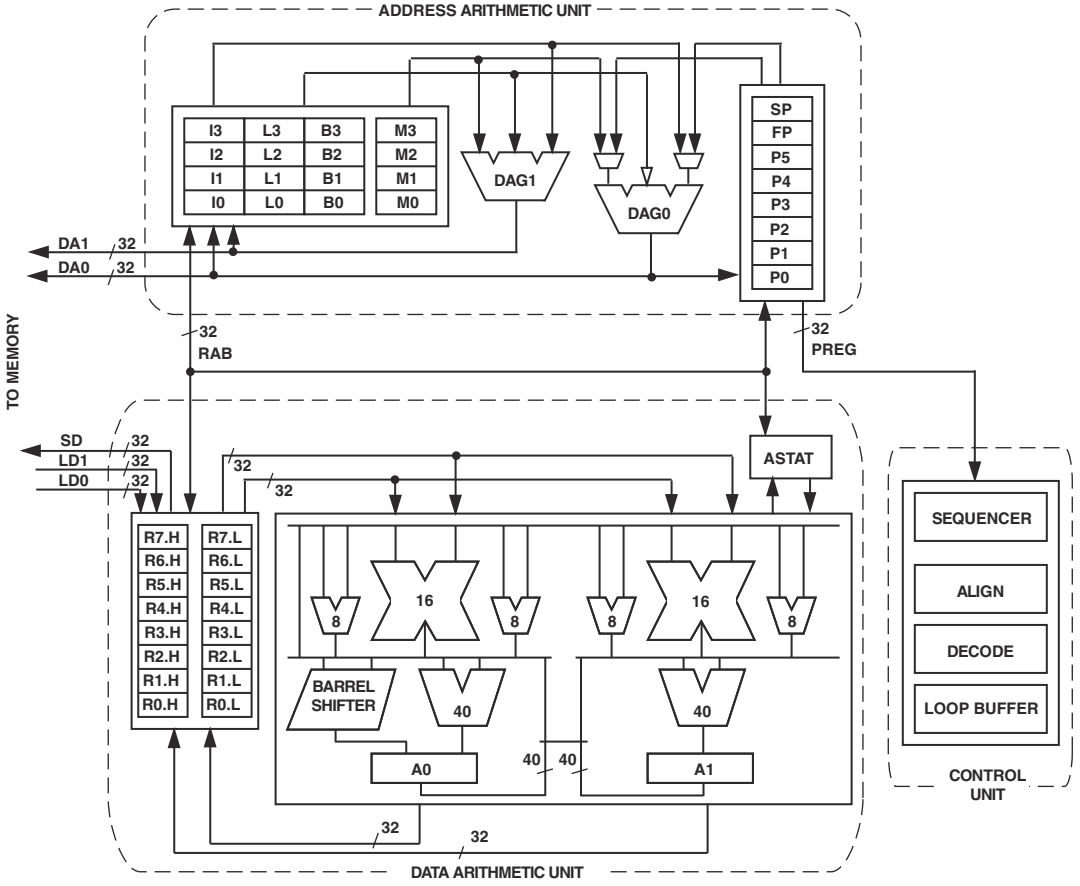


Figure 1-2. Processor Core Architecture

The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multiported register file and instruction constant fields.

Each MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. Signed and unsigned formats, rounding, and saturation are supported.

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

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

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

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

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

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

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

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

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

Memory Architecture

The Blackfin architecture structures memory as a single, unified 4 Gbyte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower-cost and lower performance off-chip memory systems. Table 1-2 on page 1-9 shows the memory allocation for the ADSP-BF539.

Table 1-2. Memory Comparison

Type of Memory	Memory size
Instruction SRAM/Cache	16 KB
Instruction SRAM	64 KB
Data SRAM/Cache	32 KB
Data SRAM	32 KB
Scratchpad	4 KB
Total	148 KB

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

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

Internal Memory

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

L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.

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

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

External Memory

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

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

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

The ADSP-BF539F4 and ADSP-BF539F8 processors include built-in flash memory (4 Mbit and 8 Mbit, respectively), which can be mapped to any of the four banks of asynchronous memory. See "On-Chip Flash Memory" on page 18-25 for details.

I/O Memory Space

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

Event Handling

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

- Emulation

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

- Reset

  Resets the processor.

- Nonmaskable Interrupt (NMI)

  The software watchdog timer or the $\overline{\texttt{NMI}}$ input signal to the processor generates this event. The NMI event is frequently used as a power-down indicator to initiate an orderly shutdown of the system.

- Exceptions

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

- Interrupts

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

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

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

Core Event Controller (CEC)

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

System Interrupt Controller (SIC)

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

DMA Support

The ADSP-BF539 processor has multiple, independent DMA controllers that support automated data transfers with minimal overhead for the processor core. DMA transfers can occur between the ADSP-BF539 processor internal memories and any of its DMA capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller. DMA capable peripherals include the SPORTs, SPI ports, UARTs, and PPI. Each individual DMA capable peripheral has at least one dedicated DMA channel. In addition, the MXVR peripheral has its own dedicated DMA controller, which supports its own unique set of operating modes.

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

The 2D DMA capability supports arbitrary row and column sizes up to 64K elements by 64K elements, and arbitrary row and column step sizes up to +/- 32K elements. Furthermore, the column step size can be less than the row step size, allowing implementation of interleaved data

streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.

Examples of DMA types supported include:

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

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

External Bus Interface Unit

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

PC133 SDRAM Controller

The SDRAM controller provides an interface to a single bank of industry-standard SDRAM devices or DIMMs. Fully compliant with the PC133 SDRAM standard, the bank can be configured to contain between 16 and 128 Mbytes of memory.

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

Asynchronous Controller

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

Parallel Peripheral Interface

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

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

General Purpose Mode Descriptions

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

- Input Mode - Frame Syncs and data are inputs into the PPI.
- Frame Capture Mode - Frame Syncs are outputs from the PPI, but data are inputs.
- Output Mode - Frame Syncs and data are outputs from the PPI.

Input Mode

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

Frame Capture Mode

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

Output Mode

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

ITU-R 656 Mode Descriptions

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

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

Active Video Only Mode

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

Vertical Blanking Interval Mode

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

Entire Field Mode

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

Serial Ports (SPORTs)

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

- Bidirectional, I^2S capable operation

  Each SPORT has two sets of independent transmit and receive pins, enabling 16 channels of I^2S stereo audio.

- Buffered (8 deep) transmit and receive ports

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

- Clocking

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

- Word length

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

- 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 processor has three SPI-compatible ports (SPI0, SPI1, SPI2) that enable the processor to communicate with multiple SPI-compatible devices.

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

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

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

Timers

There are four general-purpose programmable timer units in the processor. Three timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events.

These timer units can be synchronized to an external clock input connected to the PF1 pin (TACLK), an external clock input to the PPI_CLK pin (TMRCLK), or to the internal SCLK.

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

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

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

UART Ports

The processor has three full-duplex, Universal Asynchronous Receiver/Transmitter (UART) ports (UART0, UART1, UART2), which are fully compatible with PC-standard UARTs. The UART ports provide a simplified UART interface to other peripherals or hosts, providing full-duplex, DMA-supported, asynchronous transfers of serial data. The UART ports include support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART ports support two modes of operation:

- Programmed I/O (PIO)

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

- Direct Memory Access (DMA)

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

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

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

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

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

Media Transceiver MAC Layer (MXVR)

The processor provides a Media Transceiver (MXVR) MAC layer, allowing the processor to be connected directly to a MOST® network, through just an FOT.

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

The MXVR supports synchronous data, asynchronous packets and control messages. Data is transferred to or from the synchronous channels through eight DMA engines that work autonomously from the processor core. Furthermore two DMA engines support asynchronous traffic and a further two support control message traffic.

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

The MXVR peripheral can wake up the ADSP-BF539 processors 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-BF539 processors from sleep mode and wake up the on-chip internal voltage regulator from the powered-down hibernate state. These features allow the ADSP-BF539 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. For 44.1 kHz frame syncs, use a 45.1584 MHz crystal or oscillator; for 48 kHz frame syncs, use a 49.152 MHz crystal or oscillator. If using a crystal to provide the MXVR clock, use a parallel-resonant, fundamental mode, microprocessor-grade crystal.

Controller Area Network Port

The Controller Area Network port (CAN) provides a two-wire interface for communication with other CAN compliant devices. Features of the CAN port include error detection, multi-mastering, prioritization of messages through arbitration, and a 32-entry mailbox RAM. Transfer rates typically approach 1 Mbps.

Two-Wire Interface Port

The processor has two TWI (Two-Wire Interface) ports (TWI0, TWI1) that support synchronous serial transfers over a two-wire system with I^2C compliant devices. Features include simultaneous master and slave operation, multi-master arbitration, 400 Kbps data rates, master clock synchronization, and 7-bit addressing.

Real-Time Clock

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

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

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

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

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

Watchdog Timer

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

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

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

Programmable Flags

The processor has 16 bidirectional programmable flag (PFx) pins PF[15:0]. Each pin can be individually configured using the flag control, status, and interrupt registers.

Flag direction control register – specifies the direction of each individual PFx pin as input or output.

Flag control and status registers – the processor employs a “write one to modify” mechanism that allows any combination of individual flags to be modified in a single instruction, without affecting the level of any other flags. Four control registers are provided. One register is written in order to set flag values, one register is written in order to clear flag values, one register is written in order to toggle flag values, and one register is written in order to specify any number of flag values. Reading the flag status register allows software to interrogate the sense of the flags.

Flag interrupt mask registers – the two flag interrupt mask registers allow each individual PFx pin to function as an interrupt to the processor. Similar to the two flag control registers that are used to set and clear individual flag values, one flag interrupt mask register sets bits to enable interrupt function, and the other flag interrupt mask register clears bits to disable interrupt function. PFx pins defined as inputs can be configured to generate hardware interrupts, while output PFx pins can be triggered by software interrupts.

Flag interrupt sensitivity registers – the two flag interrupt sensitivity registers specify whether individual PFx pins are level or edge sensitive and specify—if edge sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge sensitivity.

General Purpose I/O

There are up to 38 general-purpose I/O (GPIO) pins on the processor which are separate from the programmable flag pins. The GPIO pins do not have interrupt functionality associated with them. Other than that, they are programmed similarly to the programmable flag pins.

The GPIO functionality is mulitplexed with peripheral pins. By default, the peripheral function is selected. Through software, the GPIO functionality can be selected for the pin instead.

The GPIO pins may be individually selected on a pin-by-pin basis. For example, if all the pins of a SPORT are not required, the unused pins may be used as GPIO.

Clock Signals

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

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

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

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

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

The media transceiver (MXVR) peripheral has its own clock input, which is used to drive its own internal PLL.

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

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

Sleep Mode (High Dynamic Power Savings)

The sleep mode reduces dynamic power consumption by disabling the clock to the processor core. The sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external

event or RTC activity will wake 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 memory is not supported.

Deep Sleep Mode (Maximum Dynamic Power Savings)

The deep sleep mode maximizes dynamic power savings by disabling the clocks to the processor core and to all synchronous systems. Asynchronous systems, such as the RTC, may still be running but 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, assertion of the reset interrupt or the RTC asynchronous interrupt causes the processor to transition to the active mode.

Hibernate State

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

The processor can be programmed to wake up from hibernate by reset, the RTC, CAN, or MXVR. If the CAN or MXVR are unused, a general-purpose wake-up event is also supported.

Voltage Regulation

The processor provides an on-chip voltage regulator that can generate internal voltage levels from an external 2.7 V to 3.6 V supply. The regulator controls the internal logic voltage levels and is programmable with the Voltage Regulator control register (VR_CTL) in increments of 50 mV. The regulator can also be disabled and bypassed at user discretion.

Boot Modes

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

- Execute from 16-bit external memory—execution starts from address 0x2000 0000 with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from 8-bit or 16-bit external flash memory—the 8-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup). For ADSP-BF539F processors, booting from the on-chip flash is supported via asynchronous memory bank 0 if $\overline{\text{FCE}}$ is connected to $\overline{\text{AMS0}}$.
- Boot from an SPI host in SPI Slave Mode—the SPI is configured as an SPI slave device and a host is used to boot the processor.
- Boot from an 8-/16-/24-bit addressable SPI in SPI Master Mode—support for Atmel AT45DB041B, AT45DB081B, AT45D161B Data Flash® devices. The SPI uses the PF2 output pin to select a single SPI EEPROM device.

For each of the boot modes, a 10-byte header is first read from an external memory device. The header specifies the number of bytes to be transferred and the memory destination address. Multiple memory blocks may be loaded by any boot sequence. Once all blocks are loaded, program execution commences from the start of L1 instruction SRAM (0xFFA0 0000).

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

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

Instruction Set Description

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

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

Seamlessly integrated DSP/CPU features are optimized for both 8-bit and 16-bit operations.

A multi-issue load/store modified-Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle.

All registers, I/O, and memory are mapped into a unified 4G byte memory space, providing a simplified programming model.

Microcontroller features, such as arbitrary bit and bit-field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data-types; and separate user and supervisor stack pointers.

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

Development Tools

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

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

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

- Read and write core and peripheral registers
- Plot memory

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

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

2 COMPUTATIONAL UNITS

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

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

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

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

Figure 2-1 provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data

buses leads to a better understanding of proper data flow for computations. Next, details about the processor's advanced parallelism reveal how to take advantage of multifunction instructions.

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

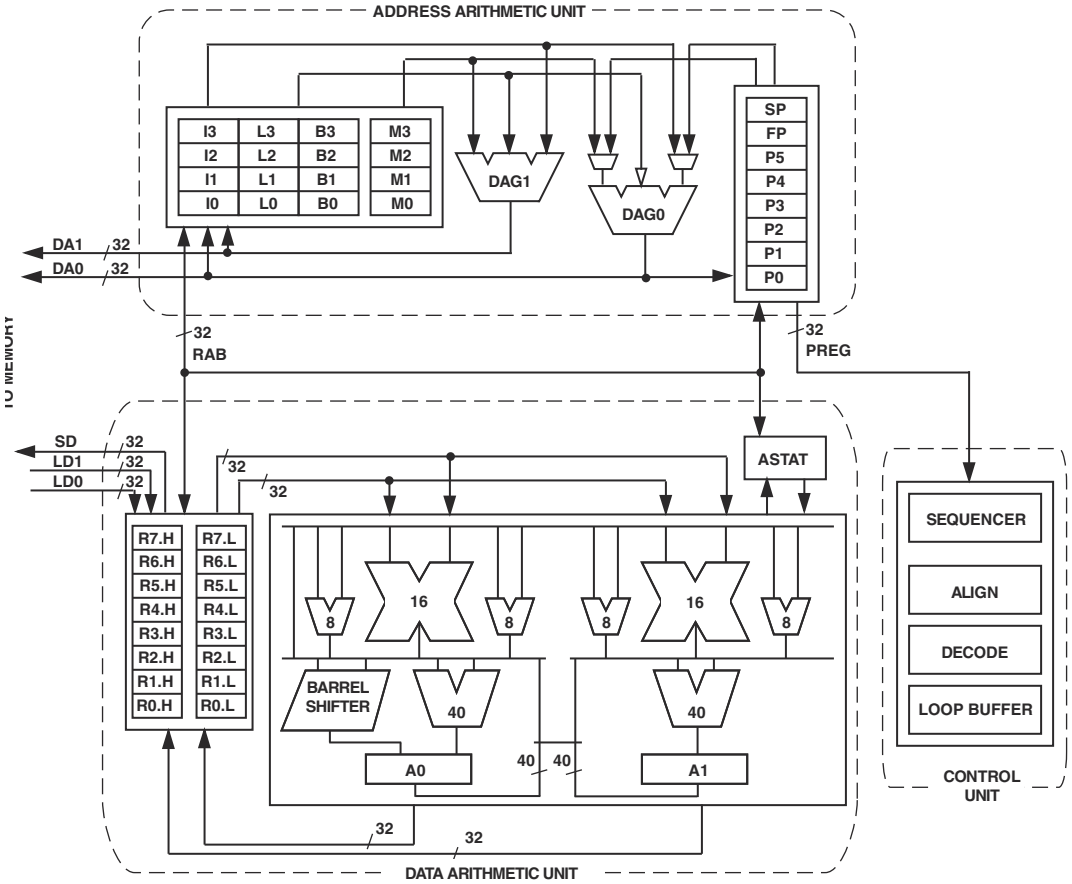


Figure 2-1. Processor Core Architecture

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

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

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

Using Data Formats

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

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

Binary String

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

Unsigned

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

Signed Numbers: Two's-Complement

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

Fractional Representation: 1.15

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

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

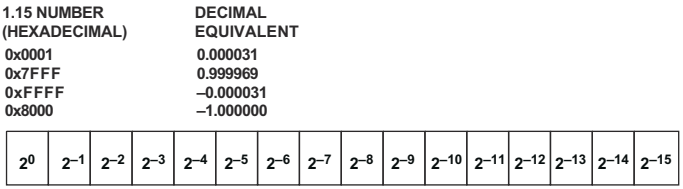


Figure 2-2. Bit Weighting for 1.15 Numbers

Register Files

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

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

For more information, see Chapter 5, "Data Address Generators."

The processor register files appear in Figure 2-3.

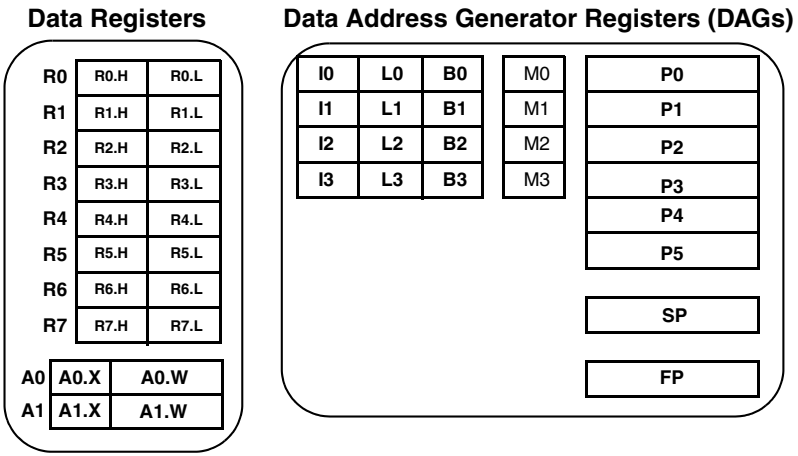


Figure 2-3. Register Files

In the processor, a word is 32 bits long; H denotes the high order 16 bits of a 32-bit register; L denotes the low order 16 bits of a 32-bit register. For example, `A0.W` contains the lower 32 bits of the 40-bit `A0` register; `A0.L` contains the lower 16 bits of `A0.W`, and `A0.H` contains the upper 16 bits of `A0.W`.

Data Register File

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

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

Accumulator Registers

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

Pointer Register File

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

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

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

DAG Register Set

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

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

All DAG registers are 32 bits wide.

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

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

Register File Instruction Summary

Table 2-1 lists the register file instructions. For more information about assembly language syntax, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

- Allreg denotes: `R[7:0]`, `P[5:0]`, `SP`, `FP`, `I[3:0]`, `M[3:0]`, `B[3:0]`, `L[3:0]`, `A0.X`, `A0.W`, `A1.X`, `A1.W`, `ASTAT`, `RETS`, `RETI`, `RETX`, `RETN`, `RETE`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `USP`, `SEQSTAT`, `SYSCFG`, `CYCLES`, and `CYCLES2`.
- A*n* denotes either ALU Result register `A0` or `A1`.
- Dreg denotes any data register file register.
- Sysreg denotes the system registers: `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `CYCLES`, and `CYCLES2`.
- Preg denotes any pointer register, `FP`, or `SP` register.
- Dreg_even denotes `R0`,`R2`,`R4`, or `R6`.

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

Table 2-1. Register File Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
allreg = allreg ; [1]	–	–	–	–	–	–	–
A*n* = A*n* ;	–	–	–	–	–	–	–
A*n* = Dreg ;	–	–	–	–	–	–	–
Dreg_even = A0 ;	*	*	–	–	–	–	*
Dreg_odd = A1 ;	*	*	–	–	–	–	*
Dreg_even = A0, Dreg_odd = A1 ;	*	*	–	–	–	–	*

Table 2-1. Register File Instruction Summary (Cont'd)

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
Dreg_odd = A1, Dreg_even = A0 ;	*	*	–	–	–	–	*
IF CC DPreg = DPreg ;	–	–	–	–	–	–	–
IF ! CC DPreg = DPreg ;	–	–	–	–	–	–	–
Dreg = Dreg_lo (Z) ;	*	**	**	–	–	–	**/–
Dreg = Dreg_lo (X) ;	*	*	**	–	–	–	**/–
A*n*.X = Dreg_lo ;	–	–	–	–	–	–	–
Dreg_lo = A*n*.X ;	–	–	–	–	–	–	–
A*n*.L = Dreg_lo ;	–	–	–	–	–	–	–
A*n*.H = Dreg_hi ;	–	–	–	–	–	–	–
Dreg_lo = A0 ;	*	*	–	–	–	–	*
Dreg_hi = A1 ;	*	*	–	–	–	–	*
Dreg_hi = A1 ; Dreg_lo = A0 ;	*	*	–	–	–	–	*
Dreg_lo = A0 ; Dreg_hi = A1 ;	*	*	–	–	–	–	*
Dreg = Dreg_byte (Z) ;	*	**	**	–	–	–	**/–
Dreg = Dreg_byte (X) ;	*	*	**	–	–	–	**/–

1 Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

Data Types

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

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

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

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

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values. For details, refer to the instructions in *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

- s = sign bit(s)
- d = data bit(s)
- "." = decimal point by convention; however, a decimal point does not literally appear in the number.
- Italics denotes data from a source other than adjacent bits.

Table 2-2. Data Formats

Format	Representation in Memory	Representation in 32-bit Register
32.0 Unsigned Word	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
32.0 Signed Word	sddd dddd dddd dddd dddd dddd dddd dddd	sddd dddd dddd dddd dddd dddd dddd dddd
16.0 Unsigned Half Word	dddd dddd dddd dddd	0000 0000 0000 0000 dddd dddd dddd dddd
16.0 Signed Half Word	sddd dddd dddd dddd	ssss ssss ssss ssss sddd dddd dddd dddd

Table 2-2. Data Formats (Cont'd)

Format	Representation in Memory	Representation in 32-bit Register
8.0 Unsigned Byte	dddd dddd	0000 0000 0000 0000 0000 0000 dddd dddd
8.0 Signed Byte	sddd dddd	ssss ssss ssss ssss ssss ssss sddd dddd
0.16 Unsigned Fraction	.dddd dddd dddd dddd	0000 0000 0000 0000 .dddd dddd dddd dddd
1.15 Signed Fraction	s.ddd dddd dddd dddd	ssss ssss ssss ssss s.ddd dddd dddd dddd
0.32 Unsigned Fraction	.dddd dddd dddd dddd dddd dddd dddd dddd	.dddd dddd dddd dddd dddd dddd dddd dddd
1.31 Signed Fraction	s.ddd dddd dddd dddd dddd dddd dddd dddd	s.ddd dddd dddd dddd dddd dddd dddd dddd
Packed 8.0 Unsigned Byte	dddd dddd *dddd dddd* dddd dddd *dddd dddd*	dddd dddd *dddd dddd* dddd dddd *dddd dddd*
Packed 0.16 Unsigned Fraction	.dddd dddd dddd dddd *.dddd dddd dddd dddd*	.dddd dddd dddd dddd *.dddd dddd dddd dddd*
Packed 1.15 Signed Fraction	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*

Endian Byte Order

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

ALU Data Types

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

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

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

ALU results generate status information. For more information about using ALU status, see "ALU Instruction Summary" on page 2-29.

Multiplier Data Types

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

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

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register (A0, A1). This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. The resulting format appears in Figure 2-4 on page 2-17.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation. This result format appears in Figure 2-5 on page 2-18.

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

Shifter Data Types

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

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

Shifter results generate status information. For more information about using shifter status, see "Shifter Instruction Summary" on page 2-55

Arithmetic Formats Summary

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

Table 2-3. ALU Arithmetic Formats

Operation	Operand Formats	Result Formats
Addition	Signed or unsigned	Interpret flags
Subtraction	Signed or unsigned	Interpret flags
Logical	Binary string	Same as operands
Division	Explicitly signed or unsigned	Same as operands

Table 2-4. Multiplier Fractional Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Addition	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Subtraction	1.15 explicitly signed or unsigned	2.30 shifted to 1.31

Table 2-5. Multiplier Arithmetic Integer Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	16.0 explicitly signed or unsigned	32.0 not shifted
Multiplication/Addition	16.0 explicitly signed or unsigned	32.0 not shifted
Multiplication/Subtraction	16.0 explicitly signed or unsigned	32.0 not shifted

Table 2-6. Shifter Arithmetic Formats

Operation	Operand Formats	Result Formats
Logical Shift	Unsigned binary string	Same as operands
Arithmetic Shift	Signed	Same as operands
Exponent Detect	Signed	Same as operands

Using Multiplier Integer and Fractional Formats

For multiply-and-accumulate functions, the processor provides two choices—fractional arithmetic for fractional numbers (1.15) and integer arithmetic for integers (16.0).

For fractional arithmetic, the 32-bit product output is format adjusted—sign-extended and shifted one bit to the left—before being added to accumulator A0 or A1. For example, bit 31 of the product lines up with bit 32 of A0 (which is bit 0 of A0.X), and bit 0 of the product lines up with bit 1 of A0 (which is bit 1 of A0.W). The least significant bit (LSB) is zero filled. The fractional multiplier result format appears in Figure 2-4.

For integer arithmetic, the 32-bit product register is not shifted before being added to A0 or A1. Figure 2-5 shows the integer mode result placement.

With either fractional or integer operations, the multiplier output product is fed into a 40-bit adder/subtracter which adds or subtracts the new product with the current contents of the A0 or A1 register to produce the final 40-bit result.

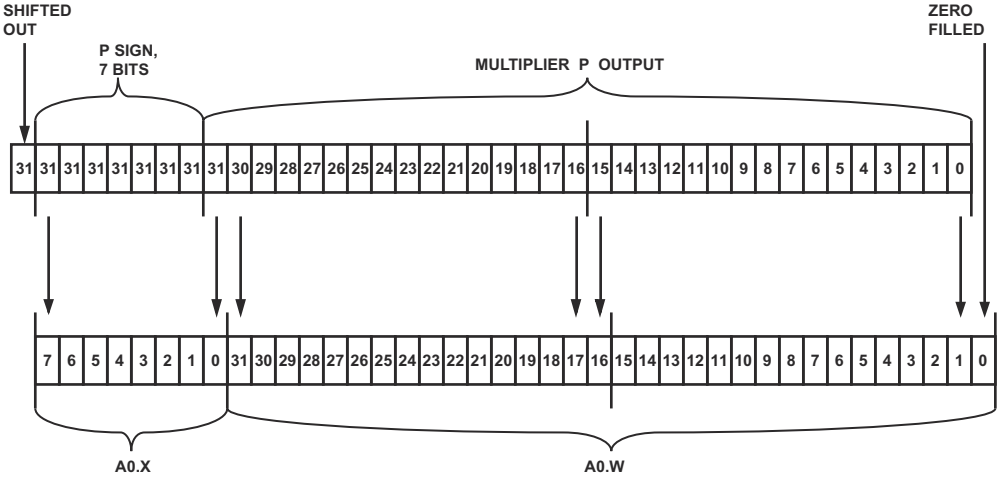


Figure 2-4. Fractional Multiplier Results Format

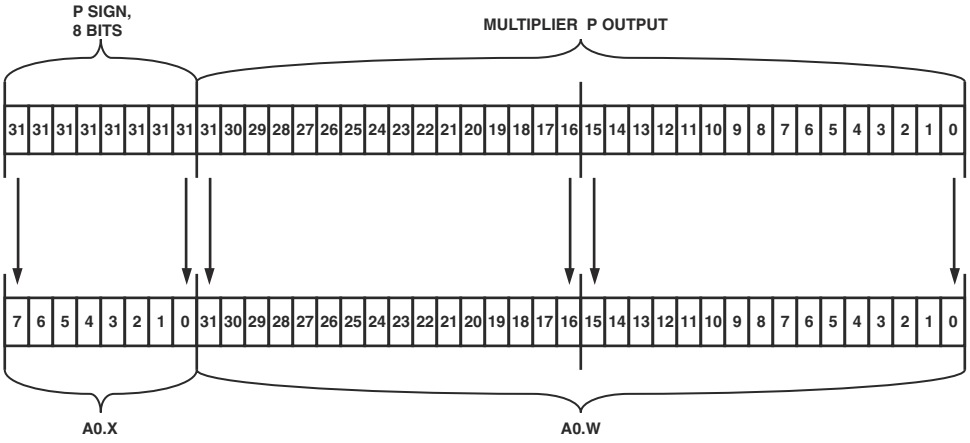


Figure 2-5. Integer Multiplier Results Format

Rounding Multiplier Results

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

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

For most algorithms, unbiased rounding is preferred.

Unbiased Rounding

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

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

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

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

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

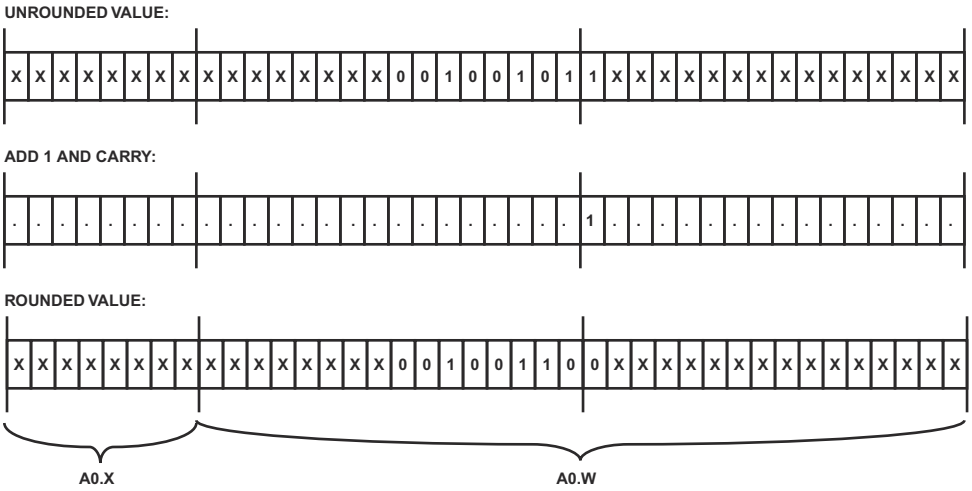


Figure 2-6. Typical Unbiased Multiplier Rounding

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

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

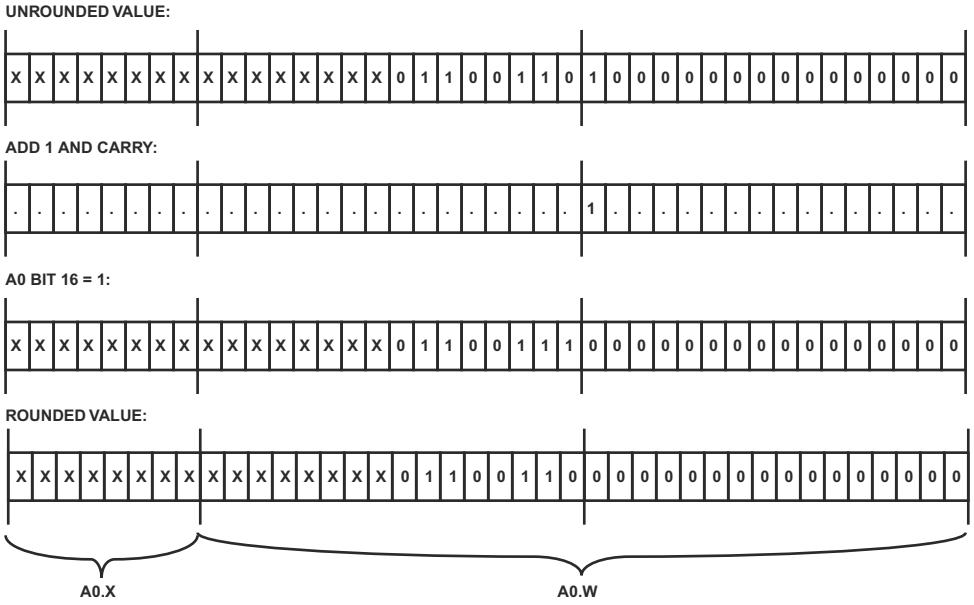


Figure 2-7. Avoiding Net Bias in Unbiased Multiplier Rounding

Biased Rounding

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

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

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

Table 2-7. Biased Rounding in Multiplier Operation

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 8000	0x00 0001 8000	0x00 0000 0000
0x00 0001 8000	0x00 0002 0000	0x00 0002 0000
0x00 0000 8001	0x00 0001 0001	0x00 0001 0001
0x00 0001 8001	0x00 0002 0001	0x00 0002 0001
0x00 0000 7FFF	0x00 0000 FFFF	0x00 0000 FFFF
0x00 0001 7FFF	0x00 0001 FFFF	0x00 0001 FFFF

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

Truncation

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

Special Rounding Instructions

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

For example:

```
R3.L = R4 (RND) ;
```

performs biased rounding at bit 16, depositing the result in a half word.

```
R3.L = R4 + R5 (RND12) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 12, depositing the result in a half word.

```
R3.L = R4 + R5 (RND20) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

Using Computational Status

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

ASTAT Register

Figure 2-8 describes the arithmetic status (ASTAT) register. The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

Arithmetic Logic Unit (ALU)

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

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

ALU Operations

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

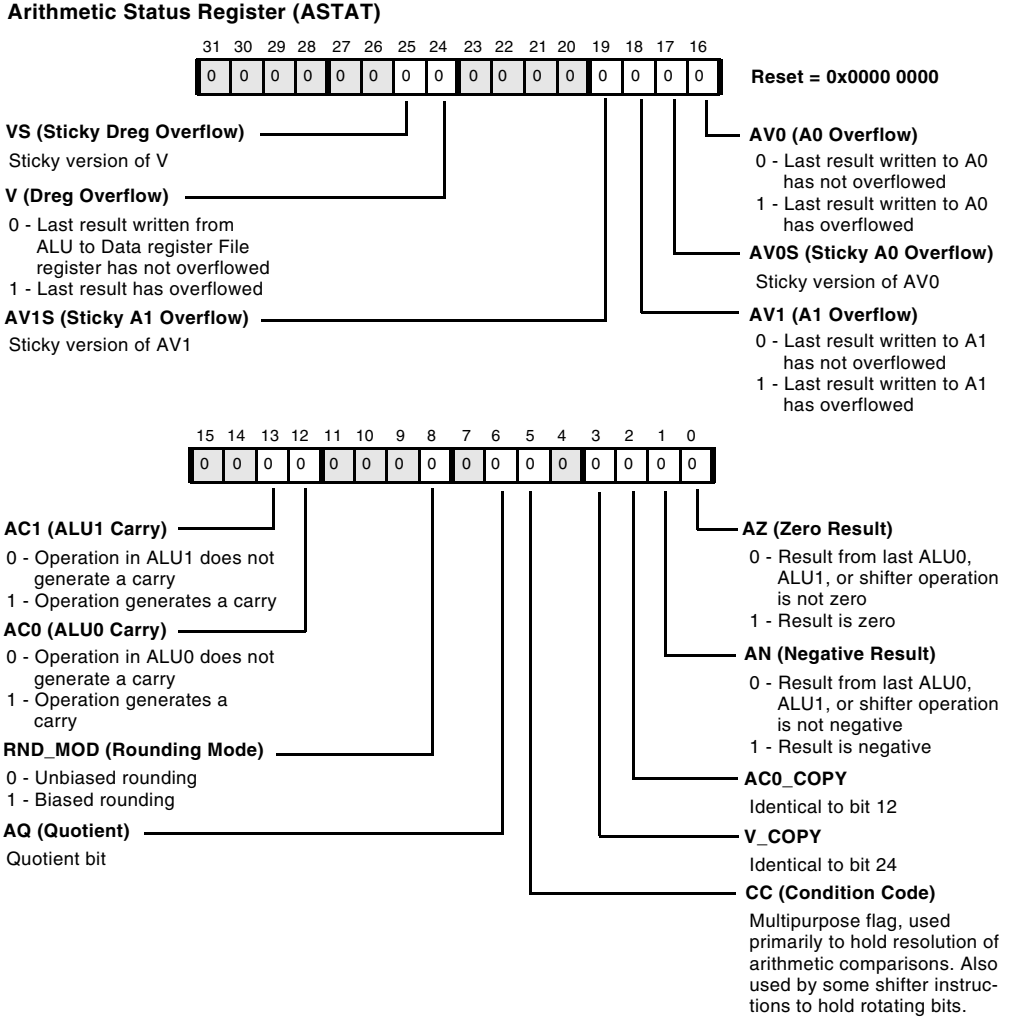


Figure 2-8. Arithmetic Status Register

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

Table 2-8. Inputs and Outputs of Each ALU

Input	Output
Two or four 16-bit operands	One or two 16-bit results
Two 32-bit operands	One 32-bit result
32-bit result from the multiplier	Combination of 32-bit result from the multiplier with a 40-bit accumulation result

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.

Single 16-Bit Operations

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

For example:

```
R3.H = R1.H + R2.L (NS) ;
```

adds the 16-bit contents of `R1.H` (`R1` high half) to the contents of `R2.L` (`R2` low half) and deposits the result in `R3.H` (`R3` high half) with no saturation.

Dual 16-Bit Operations

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

For example:

```
R3 = R1 +|- R2 (S) ;
```

adds the 16-bit contents of `R2.H` (`R2` high half) to the contents of `R1.H` (`R1` high half) and deposits the result in `R3.H` (`R3` high half) with saturation.

The instruction also subtracts the 16-bit contents of `R2.L` (`R2` low half) from the contents of `R1.L` (`R1` low half) and deposits the result in `R3.L` (`R3` low half) with saturation (see Figure 2-10 on page 2-35).

Quad 16-Bit Operations

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

For example:

```
R3 = R0 +|+ R1, R2 = R0 -|- R1 (S) ;
```

performs four operations:

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

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

Explicitly, the four equivalent instructions are:

```
R3.H = R0.H + R1.H (S) ;
R3.L = R0.L + R1.L (S) ;
R2.H = R0.H - R1.H (S) ;
R2.L = R0.L - R1.L (S) ;
```

Single 32-Bit Operations

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

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

Instructions may not intermingle pointer registers with data registers.

For example:

```
R3 = R1 + R2 (NS) ;
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation.

```
R3 = R1 + R2 (S) ;
```

adds the 32-bit contents of R1 to the 32-bit contents of R2 and deposits the result in R3 with saturation.

Dual 32-Bit Operations

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

For example:

```
R3 = R1 + R2, R4 = R1 - R2 (NS) ;
```

adds the 32-bit contents of R2 to the 32-bit contents of R1 and deposits the result in R3 with no saturation.

The instruction also subtracts the 32-bit contents of R2 from that of R1 and deposits the result in R4 with no saturation.

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

For example:

```
R3 = A0 + A1, R4 = A0 - A1 (S) ;
```

transfers to the result registers two 32-bit, saturated, sum and difference values of the ALU registers.

ALU Instruction Summary

Table 2-9 lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status flags, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

- Dreg denotes any data register file register.
- Preg denotes any pointer register, `FP`, or `SP` register.
- Dreg_lo_hi denotes any 16-bit register half in any data register File register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- imm7 denotes a signed, 7-bit wide, immediate value.
- A*n* denotes either ALU Result register `A0` or `A1`.
- DIVS denotes a divide sign primitive.
- DIVQ denotes a divide quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number, minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one, and a threshold value.

- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.
- *d* indicates `AQ` contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-9. ALU Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Preg = Preg + Preg ;	–	–	–	–	–	–	–
Preg += Preg ;	–	–	–	–	–	–	–
Preg –= Preg ;	–	–	–	–	–	–	–
Dreg = Dreg + Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg – Dreg (S) ;	*	*	*	–	–	*	–
Dreg = Dreg + Dreg, Dreg = Dreg – Dreg ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (S) ;	*	*	*	–	–	*	–
Dreg = Dreg +|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|+Dreg, Dreg = Dreg –|– Dreg ;	*	*	–	–	–	*	–

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg ;	*	*	–	–	–	*	–
Dreg = A*n* + A*n*, Dreg = A*n* – A*n* ;	*	*	*	–	–	*	–
Dreg += imm7 ;	*	*	*	–	–	*	–
Preg += imm7 ;	–	–	–	–	–	–	–
Dreg = (A0 += A1) ;	*	*	*	*	–	*	–
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	–	*	–
A0 += A1 ;	*	*	*	*	–	–	–
A0 –= A1 ;	*	*	*	*	–	–	–
DIVS (Dreg, Dreg) ;	*	*	*	*	–	–	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	–	–	d
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = ABS Dreg (V) ;	*	**	–	–	–	*	–
A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = ABS A*n*, A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = –A*n* ;	*	*	*	*	*	*	–
A*n* = –A*n*, A*n* =– A*n* ;	*	*	*	*	*	*	–
A*n* = A*n* (S) ;	*	*	–	*	*	–	–
A*n* = A*n* (S), A*n* = A*n* (S) ;	*	*	–	*	*	–	–
Dreg_lo_hi = Dreg (RND) ;	*	*	–	–	–	*	–

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo = SIGNBITS Dreg ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS Dreg_lo_hi ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS An ;	–	–	–	–	–	–	–
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V) ;	–	–	–	–	–	–	–
Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);	–	–	–	–	–	–	–
Dreg = Dreg & Dreg ;	*	*	**	–	–	**/–	–
Dreg = ~ Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg | Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg ^ Dreg ;	*	*	**	–	–	**/–	–
Dreg =– Dreg ;	*	*	*	–	–	*	–

ALU Data Flow Details

Figure 2-9 shows a more detailed diagram of the arithmetic units and the data register file, which appears in Figure 2-1 on page 2-2.

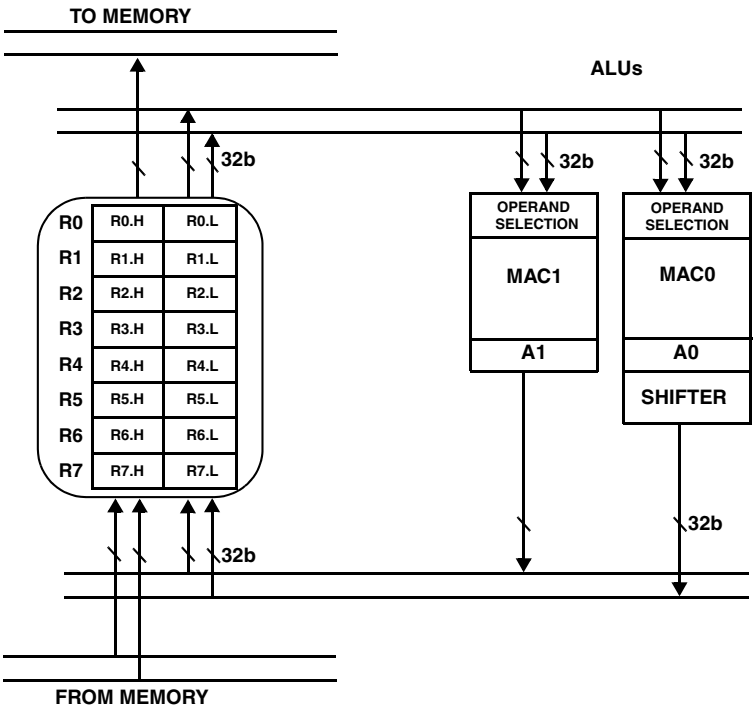


Figure 2-9. Register Files and ALUs

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

Each ALU performs 40-bit addition for the accumulation of the multiplier results, as well as 32-bit and dual 16-bit operations. Each ALU has two 32-bit input ports that can be considered a pair of 16-bit operands or a single 32-bit operand. For single 16-bit operations, any of the four possible 16-bit operands may be used with any of the other 16-bit operands presented at the input to the ALU.

As shown in Figure 2-10, for dual 16-bit operations, the high halves and low halves are paired, providing four possible combinations of addition and subtraction.

```
(A) H + H, L + L    (B) H + H, L - L
(C) H - H, L + L    (D) H - H, L - L
```

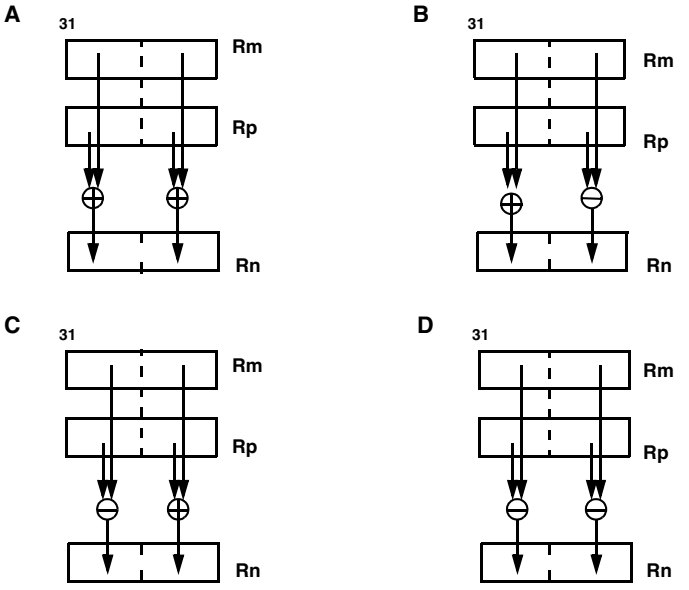


Figure 2-10. Dual 16-Bit ALU Operations

Dual 16-Bit Cross Options

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

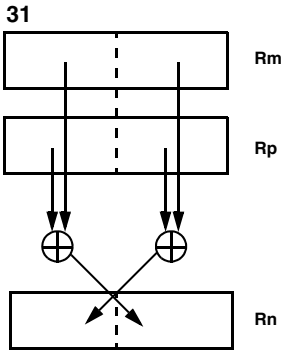


Figure 2-11. Cross Options for Dual 16-Bit ALU Operations

ALU Status Signals

Each ALU generates six status signals: the zero (AZ) status, the negative (AN) status, the carry (ACn) status, the sticky overflow (AVnS) status, the immediate overflow (AVn) status, and the quotient (AQ) status. All arithmetic status signals are latched into the arithmetic status register (ASTAT) at the end of the cycle. For the effect of ALU instructions on the status flags, see Table 2-9 on page 2-31.

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

ALU Division Support Features

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

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

Special SIMD Video ALU Operations

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

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

For more information about the operation of these instructions, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

Multiply Accumulators (Multipliers)

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

Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.

Inputs are treated as fractional or integer, unsigned or two's-complement. Multiplier instructions include:

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

Multiplier Operation

Each multiplier has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any data registers in the data register file. Each multiplier can accumulate results in its accumulator register, `A1` or `A0`. The accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

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

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction. Multiplier instruction options specify the data format of the inputs. See "Multiplier Instruction Options" on page 2-43 for more information.

Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-9 on page 2-34, each multiplier has a dedicated accumulator, `A0` or `A1`. Each accumulator register is divided into three sections—`A0.L/A1.L` (bits 15:0), `A0.H/A1.H` (bits 31:16), and `A0.X/A1.X` (bits 39:32).

When the multiplier writes to its result accumulator registers, the 32-bit result is deposited into the lower bits of the combined accumulator register, and the MSB is sign-extended into the upper eight bits of the register (`A0.X/A1.X`).

Multiplier output can be deposited not only in the `A0` or `A1` registers, but also in a variety of 16- or 32-bit data registers in the data register file.

Rounding or Saturating Multiplier Results

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

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

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

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

Saturating Multiplier Results on Overflow

The following bits in ASTAT indicate multiplier overflow status:

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

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

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

Multiplier Instruction Summary

Table 2-10 lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status flags, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

- Dreg denotes any data register file register.
- Dreg_lo_hi denotes any 16-bit register half in any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- A*n* denotes either MAC accumulator register `A0` or `A1`.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- – Indicates no effect.

Multiplier instruction options are described in "Multiplier Instruction Options" on page 2-43.

Table 2-10. Multiplier Instruction Summary

Instruction	ASTAT Status Flags		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
A*n* = Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
A*n* += Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
An –= Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	–	–	–

Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

default	No option; input data is signed fraction.
`(IS)`	Input data operands are signed integer. No shift correction is made.
`(FU)`	Input data operands are unsigned fraction. No shift correction is made.
`(IU)`	Input data operands are unsigned integer. No shift correction is made.
`(T)`	Input data operands are signed fraction. When copying to the destination half register, truncates the lower 16 bits of the accumulator contents.
`(TFU)`	Input data operands are unsigned fraction. When copying to the destination half register, truncates the lower 16 bits of the accumulator contents.
`(ISS2)`	If multiplying and accumulating to a register: Input data operands are signed integer. When copying to the destination register, accumulator contents are scaled (multiplied x2 by a one-place shift-left). If scaling produces a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value.

If multiplying and accumulating to a half register:

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

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

`(W32)` Input data operands are signed fraction with no extension bits in the accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit accumulators. For this option only, this special case applies: `0x8000 x 0x8000 = 0x7FFF`.

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

Multiplier Data Flow Details

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

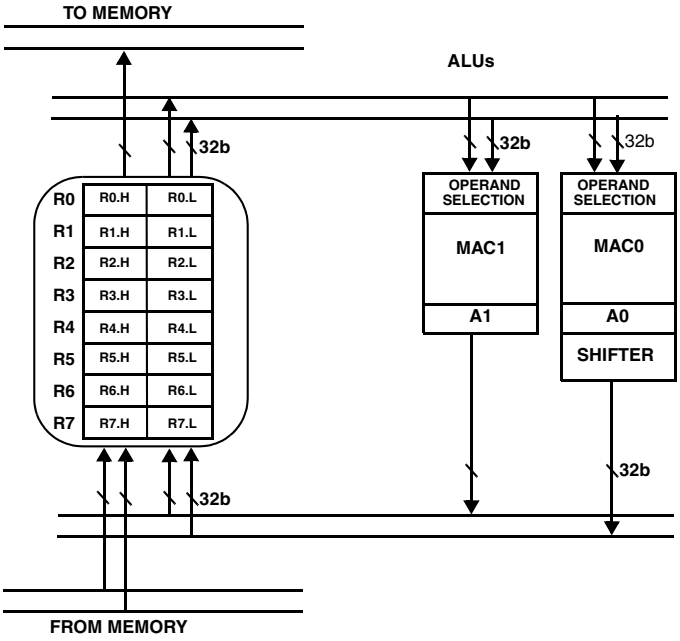


Figure 2-12. Register Files and ALUs

Each multiplier has two 16-bit inputs, performs a 16-bit multiplication, and stores the result in a 40-bit accumulator or extracts to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to chose from.

One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can contain the same register information, giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-13 show these possible combinations.

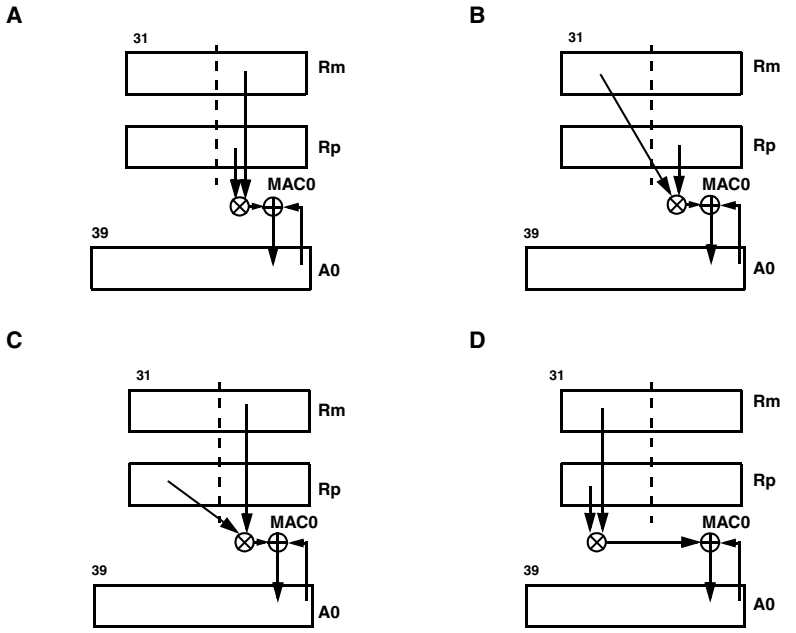


Figure 2-13. Four Possible Combinations of MAC Operations

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

Some example instructions:

```
A0 = R3.L * R4.H ;
```

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

```
A1 += R3.H * R4.H ;
```

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

Multiply Without Accumulate

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

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

- If the multiplication uses fractional operands or the IH option, then the high half of the result is extracted and stored in the 16-bit destination registers (see Figure 2-14).
- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see Figure 2-15).

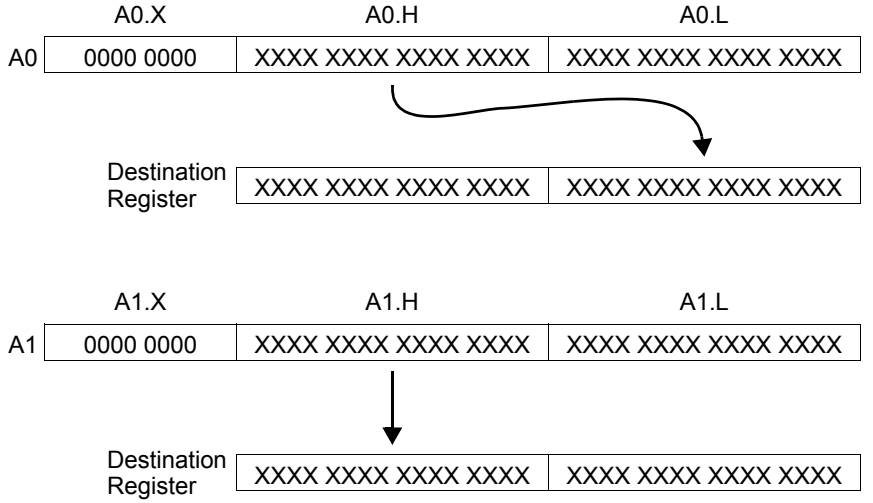


Figure 2-14. Multiplication of Fractional Operands

For example, this instruction uses fractional, unsigned operands:

```
R0.L = R1.L * R2.L (FU) ;
```

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

```
R0.H = R2.H * R3.H (IU) ;
```

The instruction deposits the lower 16 bits of the multiply answer with any required saturation into the high half of `R0`, using MAC1.

```
R0 = R1.L * R2.L ;
```

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into `R0`, using MAC0.

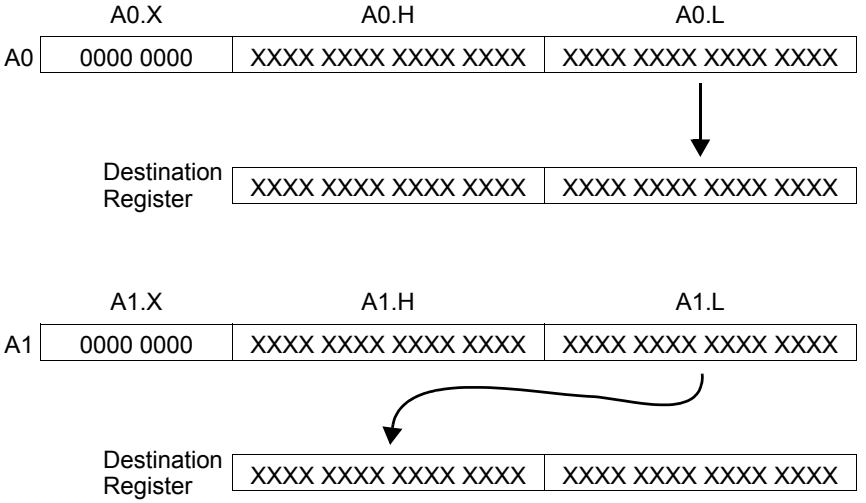


Figure 2-15. Multiplication of Integer Operands

Special 32-Bit Integer MAC Instruction

The processor supports a multi cycle 32-bit MAC instruction:
`Dreg *= Dreg`

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

The instruction takes multiple cycles to execute. Refer to the product data sheet and *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for more information about the exact operation of this instruction. This macro function is interruptable and does not modify the data in either accumulator register `A0` or `A1`.

Dual MAC Operations

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

An example of a dual multiply and accumulate instruction is

```
A1 += R1.H * R2.L, A0 += R1.L * R2.H ;
```

This instruction represents two multiply and accumulate operations.

- In one operation (MAC1) the high half of `R1` is multiplied by the low half of `R2` and added to the contents of the `A1` accumulator.
- In the second operation (MAC0) the low half of `R1` is multiplied by the high half of `R2` and added to the contents of `A0`.

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

For example:

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

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

The operand type determines the correct bits to extract from the accumulator and deposit in the 16-bit destination register. See “Multiply Without Accumulate” on page 2-47.

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

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

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

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

Barrel Shifter (Shifter)

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

Shifter Operations

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

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values (as would occur in many DSP algorithms) can use the instructions `ASHIFT` and `LSHIFT`. These are typically three-operand instructions.

Instructions that are intended to operate on a 32-bit register value and use two operands, such as instructions frequently used by a compiler, can use the `>>>` and `>>` instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter related instructions, see "Shifter Instruction Summary" on page 2-55.

Two-Operand Shifts

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

Immediate Shifts

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

The following example shows the input value downshifted.

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

results in

```
R0 contains 0000 0B6A ;
```

The following example shows the input value upshifted.

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

results in

```
R0 contains 000B 6A30 ;
```

Register Shifts

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

The following example shows the input value upshifted.

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

results in

```
R0 contains 000B 6A30 ;
```

Three-Operand Shifts

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

Immediate Shifts

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

The following example shows the input value downshifted.

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

results in

```
R1 contains 0000 0B6A ;
```

The following example shows the input value upshifted.

```
R0.L contains B6A3 ;
R1.H = R0.L << 0x04 ;
```

results in

```
R1.H contains 6A30 ;
```

Register Shifts

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

The following example shows the input value upshifted.

```
R0 contains 0000 B6A3 ;
R2.L contains 0004 ;
R1 = R0 ASHIFT by R2.L ;
```

results in

```
R1 contains 000B 6A30 ;
```

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

```
R0 contains ABCD EF12 ;
R2.L contains 0004 ;
R1 = R0 ROT by R2.L ;
```

results in

```
R1 contains BCDE F125 ;
```

Note the CC bit is included in the result, at bit 3.

Bit Test, Set, Clear, Toggle

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

The following examples show a variety of operations.

```
BITCLR ( R0, 6 ) ;
BITSET ( R2, 9 ) ;
BITTGL ( R3, 2 ) ;
CC = BITTST ( R3, 0 ) ;
```

Field Extract and Field Deposit

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

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

Shifter Instruction Summary

Table 2-11 lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status flags, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

- Dreg denotes any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.

- * 0 Indicates versions of the instruction that send results to accumulator A0 set or clear AV0.
- * 1 Indicates versions of the instruction that send results to accumulator A1 set or clear AV1.
- ** Indicates the flag is cleared.
- *** Indicates CC contains the latest value shifted into it.
- – Indicates no effect.

Table 2-11. Shifter Instruction Summary

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
BITSET (Dreg, uimm5) ;	**	*	**	–	–	–	**/–
BITTGL (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
CC = BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
CC = !BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
BITMUX (Dreg, Dreg, A0) ;	–	–	–	–	–	–	–
Dreg_lo = ONES Dreg ;	–	–	–	–	–	–	–
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	–	–	–	–	–	–	–
Dreg >>>= uimm5 ;	*	*	–	–	–	–	**/–
Dreg >>= uimm5 ;	*	*	–	–	–	–	**/–

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

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg <<= uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >>> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm5 ;	*	*	–	–	–	–	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm4 (V) ;	*	*	–	–	–	–	*
A*n* = A*n* >>> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* >> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* << uimm5 ;	*	*	–	* 0	* 1	–	–
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	–	–	–	–	*
Dreg >>>= Dreg ;	*	*	–	–	–	–	**/–
Dreg >>= Dreg ;	*	*	–	–	–	–	**/–
Dreg <<= Dreg ;	*	*	–	–	–	–	**/–
Dreg = ASHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	*
Dreg = LSHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	**/–
Dreg = ROT Dreg BY imm6 ;	–	–	–	–	–	***	–
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	*

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

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	*
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	**/–
A*n* = A*n* ASHIFT BY Dreg _lo ;	*	*	–	* 0	* 1	–	–
A*n* = A*n* ROT BY imm6 ;	–	–	–	–	–	***	–
Preg = Preg >> 1 ;	–	–	–	–	–	–	–
Preg = Preg >> 2 ;	–	–	–	–	–	–	–
Preg = Preg << 1 ;	–	–	–	–	–	–	–
Preg = Preg << 2 ;	–	–	–	–	–	–	–
Dreg = (Dreg + Dreg) << 1 ;	*	*	*	–	–	–	*
Dreg = (Dreg + Dreg) << 2 ;	*	*	*	–	–	–	*
Preg = (Preg + Preg) << 1 ;	–	–	–	–	–	–	–
Preg = (Preg + Preg) << 2 ;	–	–	–	–	–	–	–
Preg = Preg + (Preg << 1) ;	–	–	–	–	–	–	–
Preg = Preg + (Preg << 2) ;	–	–	–	–	–	–	–

3 OPERATING MODES AND STATES

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

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

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

The processor mode is determined by the Event Controller. When servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in Supervisor mode. When servicing an emulation event, the processor is in Emulation mode. When not servicing any events, the processor is in User mode.

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

MMRs cannot be read while the processor is in User mode.

Table 3-1. Identifying the Current Processor Mode

Event	Mode	IPEND
Interrupt	Supervisor	≥ 0x10 but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0.
Exception	Supervisor	≥ 0x08 The core is processing an exception event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 0, IPEND[3] = 1, and IPEND[15:4] are 0's or 1's.
NMI	Supervisor	≥ 0x04 The core is processing an NMI event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 1, and IPEND[15:2] are 0's or 1's.
Reset	Supervisor	= 0x02 As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in Supervisor mode.
Emulation	Emulator	= 0x01 The processor is in Emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1].
None	User	= 0x00

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

- Idle state
- Reset state

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

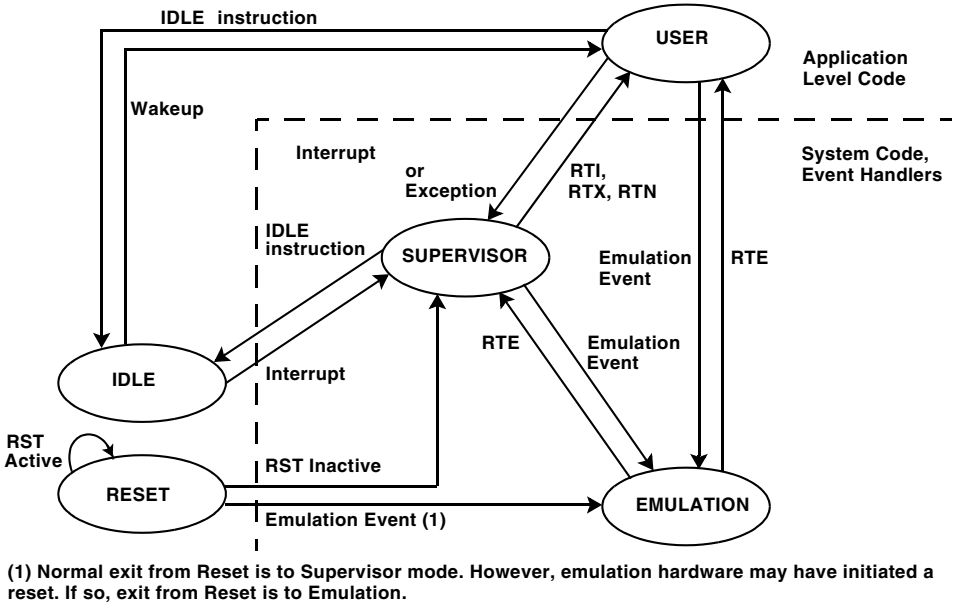


Figure 3-1. Processor Modes and States

User Mode

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

Table 3-2. Registers Accessible in User Mode

Processor Registers	Register Names
Data Registers	R[7:0], A[1:0]
Pointer Registers	P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]
Sequencer and register Registers	RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2

Protected Resources and Instructions

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

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

Table 3-3. Protected Instructions

Instruction	Description
RTI	Return from Interrupt
RTX	Return from Exception
RTN	Return from NMI
CLI	Disable Interrupts
STI	Enable Interrupts
RAISE	Force Interrupt/Reset
RTE	Return from Emulation Causes an exception only if executed outside Emulation mode

Protected Memory

Additional memory locations can be protected from User mode access. A Cacheability Protection Lookaside Buffer (CPLB) entry can be created and enabled. See "Memory Management Unit" on page 6-44 for further information.

Entering User Mode

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

Example Code to Enter User Mode Upon Reset

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

Listing 3-1. Entering User Mode from Reset

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

START :   /* Place user code here */
```

Return Instructions That Invoke User Mode

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

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

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

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

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

Current Process Activity	Return Instruction to Use	Execution Resumes at Address in This Register
Interrupt Service Routine	RTI	RETI
Exception Service Routine	RTX	RETX
Nonmaskable Interrupt Service Routine	RTN	RETN
Emulation Service Routine	RTE	RETE

Supervisor Mode

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

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured and enabled. See "Memory Management Unit" on page 6-44 for a further description. Only Supervisor mode can use the register alias `USP`, which references the User Stack Pointer in memory. This register alias is necessary because in Supervisor mode, `SP` refers to the kernel stack pointer rather than to the user stack pointer.

Normal processing begins in Supervisor mode from the Reset state. De-asserting the $\overline{\texttt{RESET}}$ signal switches the processor from the Reset state to Supervisor mode where it remains until an emulation event or Return instruction occurs to change the mode. Before the Return instruction is issued, the `RETI` register must be loaded with a valid return address.

Non-OS Environments

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

One way of keeping the processor in Supervisor mode and still allowing lower priority events to be processed is to set up and force the lowest priority interrupt (`IVG15`). Events and interrupts are described further in "Events and Sequencing" on page 4-17. After the low priority interrupt has been forced using the `RAISE 15` instruction, `RETI` can be loaded with a return address that points to user code that can execute until `IVG15` is issued. After `RETI` has been loaded, the `RTI` instruction can be issued to return from the reset event.

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

Example Code for Supervisor Mode Coming Out of Reset

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

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

```
P0.L = LO(EVT15) ;   /* Point to IVG15 in Event Vector Table */
P0.H = HI(EVT15) ;
P1.L = START ;   /* Point to start of User code */

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

P0.L = LO(IMASK) ;

R0 = [P0] ;
R1.L = EVT_IVG15 & 0xFFFF ;

R0 = R0 | R1 ;
[P0] = R0 ;   /* Enable IVG15 bit in Interrupt Mask register */

RAISE 15 ;   /* Invoke IVG15 interrupt */
P0.L = WAIT_HERE ;
P0.H = WAIT_HERE ;
RETI = P0 ;   /* RETI loaded with return address */
RTI ;   /* Return from Reset Event */
WAIT_HERE :   /* Wait here till IVG15 interrupt is serviced */
```

```
JUMP WAIT_HERE ;

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

Emulation Mode

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

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

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

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

Idle State

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

The processor remains in the Idle state until a peripheral or external device, such as a SPORT or the Real-Time Clock (RTC), generates an interrupt that requires servicing.

In Listing 3-3, core interrupts are disabled and the `IDLE` instruction is executed. When all the pending processes have completed, the core disables its clocks. Since interrupts are disabled, Idle state can be terminated only by asserting a wake up signal. For more information, see "System Interrupt Wakeup-Enable (SIC_IWRx) Registers" on page 4-26. While not required, an interrupt could also be enabled in conjunction with the wake up signal.

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

Example Code for Transition to Idle State

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

Listing 3-3. Transitioning to Idle State

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

Reset State

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

The processor remains in the Reset state as long as external logic asserts the external $\overline{\texttt{RESET}}$ signal. Upon de-assertion, the processor completes the reset sequence and switches to Supervisor mode, where it executes code found at the reset event vector.

Software in Supervisor or Emulation mode can invoke the Reset state without involving the external $\overline{\texttt{RESET}}$ signal. This can be done by issuing the Reset version of the `RAISE` instruction.

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

Table 3-5. Processor State Upon Reset

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

System Reset and Power-up

Table 3-6 describes the five types of resets. Note all resets, except System Software, reset the core.

Table 3-6. Resets

Reset	Source	Result
Hardware Reset	The $\overline{\text{RESET}}$ pin causes a hardware reset.	Resets both the core and the peripherals, including the Dynamic Power Management Controller (DPMC). Resets the No Boot on Software Reset bit in SYSCR. For more information, see "SYSCR Register" on page 3-14.
System Software Reset	Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC0 0100 causes a System Software reset.	Resets only the peripherals, excluding the RTC (Real-Time Clock) block and most of the DPMC. The DPMC resets only the No Boot on Software Reset bit in SYSCR. Does not reset the core. Does not initiate a boot sequence.
Watchdog Timer Reset[1]	Programming the watchdog timer appropriately causes a Watchdog Timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The Software Reset register (SWRST) can be read to determine whether the reset source was the watchdog timer.
Core Double-Fault Reset	If the core enters a double-fault state, and the Core Double Fault Reset Enable bit (DOUBLE_FAULT) is set in the SWRST register, then a software reset occurs.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST register can be read to determine whether the reset source was Core Double Fault.
Core-Only Software Reset	This reset is caused by executing a RAISE1 instruction or by setting the Software Reset (SYSRST) bit in the core Debug control register (DBGCTL) via emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Resets only the core. The peripherals do not recognize this reset.

1 When in Sleep or Deep Sleep modes, the Watchdog Timer Reset is not functional.

Hardware Reset

The processor chip reset is an asynchronous reset event. The RESET input pin must be de-asserted to perform a hardware reset. For more information, see *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

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

The BMODE[1:0] pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DD} or V_{SS}. The pins and the corresponding bits in SYSCR configure the Boot mode that is employed after hardware reset or System Software reset. See "Reset" on page 4-44, and Table 4-11 on page 4-47 for further information.

SYSCR Register

The values sensed from the BMODE[1:0] pins are latched into the System Reset Configuration register (SYSCR) upon the de-assertion of the RESET pin. The values are made available for software access and modification after the hardware reset sequence. Software can modify only the No Boot on Software Reset bit.

The various configuration parameters are distributed to the appropriate destinations from SYSCR (see Figure 3-2).

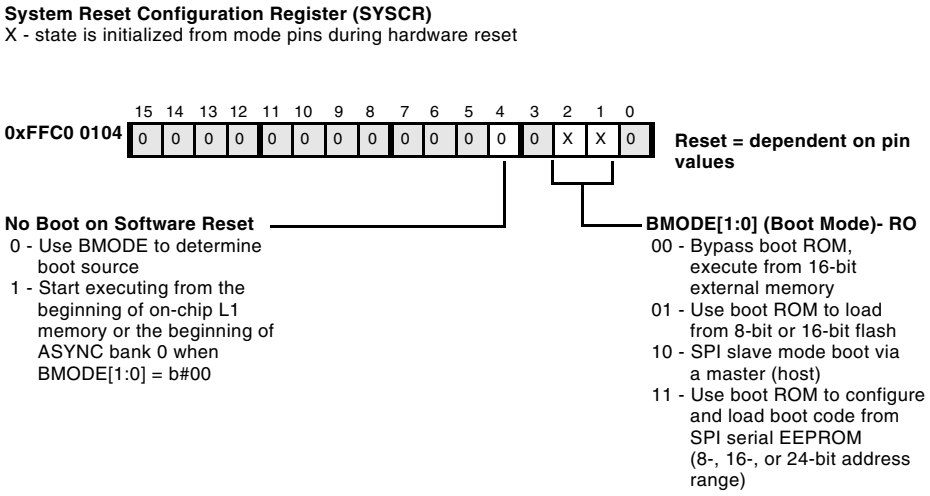


Figure 3-2. System Reset Configuration Register

Software Resets and Watchdog Timer

A software reset may be initiated in three ways:

- By the watchdog timer, if appropriately configured
- By setting the System Software Reset field in the Software Reset register (see "Software Reset Register" on page 3-17)
- By the RAISE 1 instruction

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

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

When L1 instruction memory is configured as cache, make sure the System Software reset sequence has been read into the cache.

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

For a reset generated by the watchdog timer, the processors transitions into the Boot mode sequence. The Boot mode is configured by the state of the BMODE and the No Boot on Software Reset control bits. Because the watchdog reset only resets part of the DPMC, the reset event cannot be used if the processor is in the Sleep or Deep Sleep modes of operation.

If the No Boot on Software Reset bit in SYSCR is cleared, the reset sequence is determined by the BMODE[1:0] control bits.

SWRST Register

A software reset can be initiated by setting the System Software Reset field in the Software Reset register (SWRST). Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 and Bit 13, respectively, indicate whether the Software Watchdog Timer or a Core Double Fault has generated a software reset. Bits [15:13] are read-only and cleared when the register is read. Bits [3:0] are read/write.

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

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

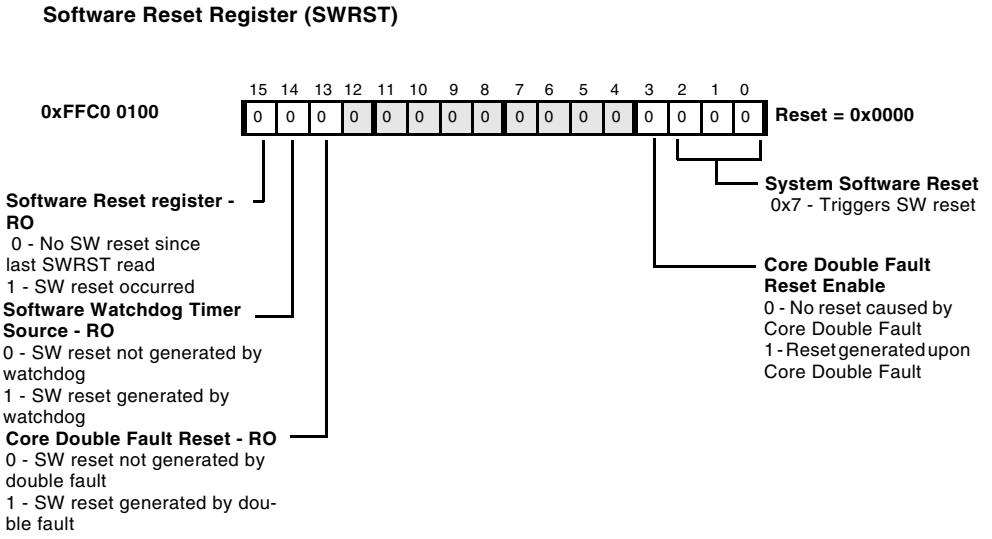


Figure 3-3. Software Reset Register

Core-Only Software Reset

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

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

Core and System Reset

To perform a system and core reset, use the code sequence shown in Listing 3-4.

Listing 3-4. Core and System Reset

```
/* Issue system soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0007 ;
W[P0] = R0 ;
SSYNC ;
/* Wait for System reset to actually reset, needs to be
5 SCLKs */
/* Assume CCLK:SCLK ratio is worst case (15:1), and use 5*15 */
P1 = 75;
LSETUP(start, end) LC0=P1;
start:
end:
  NOP;

/* Clear system soft reset */
R0.L = 0x0000 ;
W[P0] = R0 ;
SSYNC ;
/* Core reset - forces reboot */
RAISE 1 ;
```

Booting Methods

The internal boot ROM includes a small boot kernel that can either be bypassed or used to load user code from an external memory device. See Table 4-10 on page 4-44 for further information. The boot kernel reads the `BMODE[1:0]` pin state at reset to identify the download source (see Table 4-7 on page 4-23). When in boot mode 0, the processor is set to execute from 16-bit wide external memory at address 0x2000 0000 (ASYNC bank 0).

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

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

If booting from the Serial Peripheral Interface (SPI0), programmable flag pin 2 (PF2) is used as the SPI-chip select. This line must be connected for proper operation.

A core-only software reset also vectors the core to the boot ROM. Only the core is reset with the core-only software reset; this reset does not affect the rest of the system. The boot ROM kernel detects a no boot on software reset condition in SYSCR to avoid initiating a download. If this bit is set on a software reset, the processor skips the normal boot sequence and jumps to the beginning of L1 memory and begins execution.

The boot kernel assumes these conditions for the Flash Boot mode (BMODE[1:0] = b#01):

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

For SPI master mode boot (BMODE[1:0] = b#11), the boot kernel assumes that the SPI0 baud rate is 500 kHz. SPI serial EEPROMs that are 8-bit, 16-bit, and 24-bit addressable are supported. The SPI uses the PF2 output pin to select a single SPI EEPROM device. The SPI0 controller submits successive read commands at addresses 0x00, 0x0000, and 0x000000 until a valid 8-, 16-, or 24-bit addressable EEPROM is detected. It then begins clocking data into the beginning of L1 instruction memory.

The MISO0 pin must be pulled high for SPI master mode booting (BMODE[1:0] = b#11).

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

For SPI slave mode boot (BMODE[1:0] = b#10), the hardware configuration shown in Figure 3-4 is assumed.

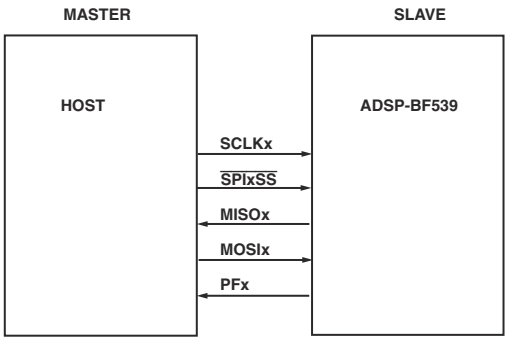


Figure 3-4. SPI Slave Boot Mode

The user-defined programmable flag PFx is an output on the Blackfin processor and an input on the host device. This flag allows the processor to hold off the host device from sending data during certain sections of the boot process. When this flag is de-asserted, the host can continue to send bytes to the processor.

4 PROGRAM SEQUENCER

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

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in Figure 2-1 on page 2-2. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps.** Program flow transfers permanently to another part of memory.
- **Interrupts and Exceptions**. A runtime event or instruction triggers the execution of a subroutine.
- **Idle.** An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

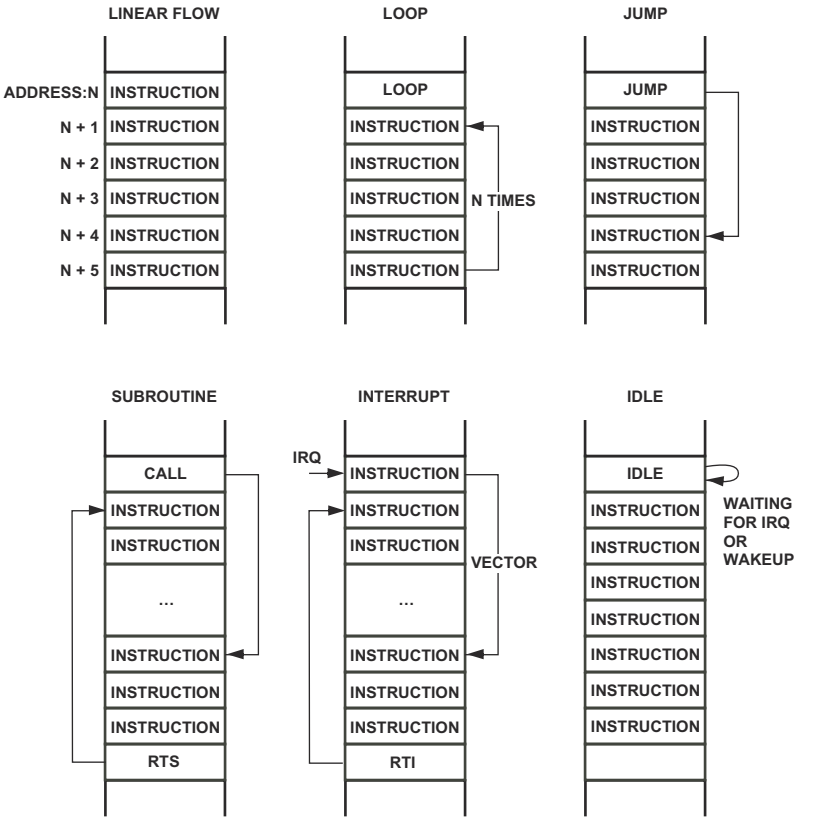


Figure 4-1. Program Flow Variations

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute.

The fetched address enters the instruction pipeline, ending with the program counter (PC). The pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The PC couples with the `RETn` registers, which store return addresses. All addresses generated by the sequencer are 32-bit memory instruction addresses.

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

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

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

Sequencer Related Registers

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

Table 4-1. Sequencer Related Registers

Register Name	Description
SEQSTAT	Sequencer status register
RETX RETN RETI RETE RETS	Return Address registers: See “Events and Sequencing” on page 4-17. Exception Return NMI Return Interrupt Return Emulation Return Subroutine Return
LC0, LC1 LT0, LT1 LB0, LB1	Zero-Overhead Loop registers: Loop Counters Loop Tops Loop Bottoms
FP, SP	Frame Pointer and Stack Pointer: see Chapter 5, “Data Address Generators”
SYSCFG	System Configuration register
CYCLES, CYCLES2	Cycle Counters: see Chapter 23, “Blackfin Processor Debug”
PC	Program Counter

Sequencer Status (SEQSTAT) Register

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

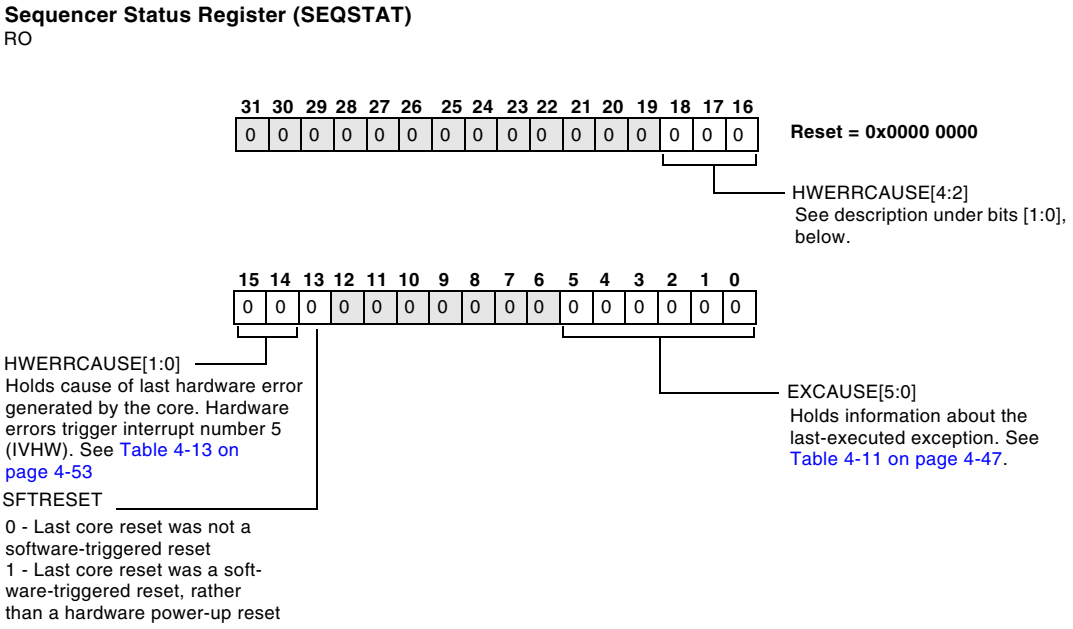


Figure 4-2. Sequencer Status Register

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

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

Table 2-2 on page 2-5 describes the 32-bit loop register sets:

Table 4-2. Loop Registers

Registers	Description	Function
LC0, LC1	Loop Counters	Maintain a count of the remaining iterations of the loop
LT0, LT1	Loop Tops	Hold the address of the first instruction within a loop
LB0, LB1	Loop Bottoms	Hold the address of the last instruction of the loop

System Configuration (SYSCFG) Register

The System Configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from the Supervisor mode.

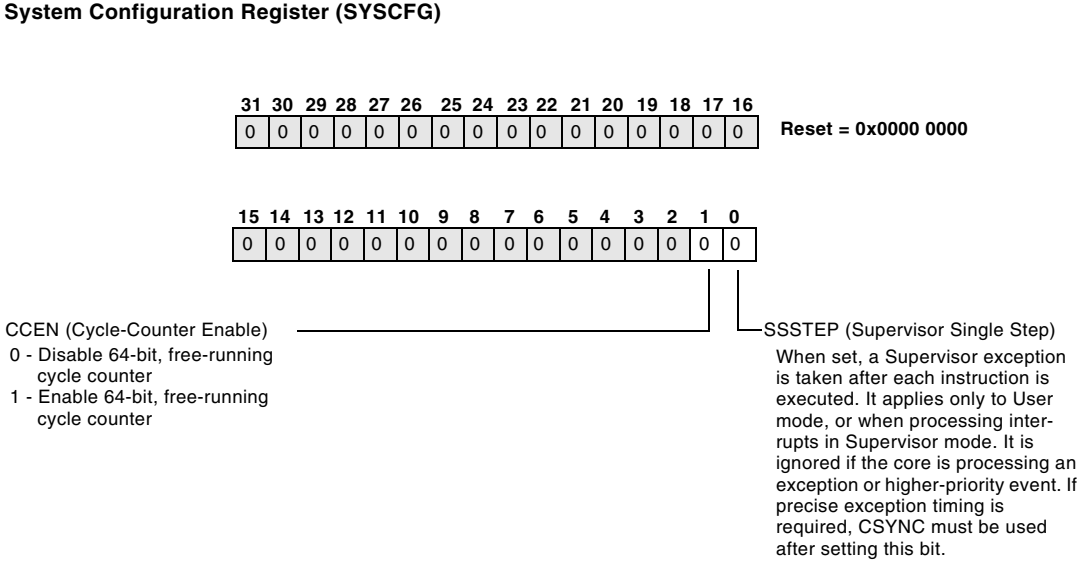


Figure 4-3. System Configuration Register

Instruction Pipeline

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

The processor has a ten-stage instruction pipeline.

Table 4-3. Stages of Instruction Pipeline

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Start instruction memory access.
Instruction Fetch 2 (IF2)	Intermediate memory pipeline.
Instruction Fetch 3 (IF3)	Finish L1 instruction memory access.
Instruction Decode (DEC)	Align instruction, start instruction decode, and access Pointer register file.
Address Calculation (AC)	Calculate data addresses and branch target address.
Execute 1 (EX1)	Start access of data memory.
Execute 2 (EX2)	register file read.
Execute 3 (EX3)	Finish accesses of data memory and start execution of dual cycle instructions.
Execute 4 (EX4)	Execute single cycle instructions.
Write Back (WB)	Write states to Data and Pointer register files and process events.

The following figure shows a diagram of the pipeline.

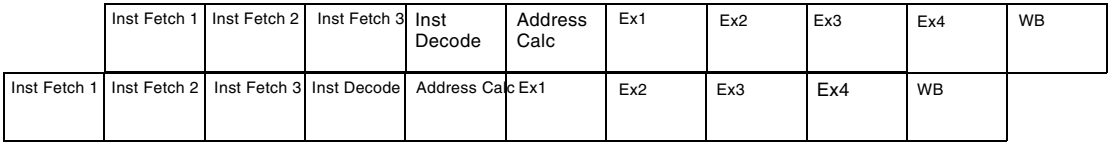


Figure 4-4. Processor Pipeline

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

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

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

The sequencer holds the fetch address until it receives a request from the alignment logic or until a change of flow occurs. It always increments the previous fetch address by 8 (the next 8 bytes). If a change of flow occurs, such as a branch or an interrupt, the sequencer communicates it to the instruction memory unit, which invalidates the data in the alignment unit.

The execution unit contains two 16-bit multipliers, two 40-bit ALUs, two 40-bit accumulators, one 40-bit shifter, a video unit (which adds 8-bit ALU support), and an 8-entry 32-bit Data register File.

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

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

Jumps

Subroutine calls and returns

Interrupts and returns

Loops

Branches and Sequencing

One type of nonsequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL` instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the `JUMP` and `CALL` instructions, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. Briefly:

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

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

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

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

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

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

Direct Short and Long Jumps

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

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (bit zero must be a zero). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.

If, at the time of writing the program, the destination is known to be less than a 13-bit offset from the current PC value, then the `JUMP.S 0xnnnn` instruction may be used. If the destination requires more than a 13-bit offset, then the `JUMP.L 0xnnnnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnnnnn`. Upon disassembly, the instruction is replaced by the appropriate `JUMP.S` or `JUMP.L` instruction.

Direct Call

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

Indirect Branch and Call

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

For example:

```
JUMP (P3) ;

CALL (P0) ;
```

PC-Relative Indirect Branch and Call

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

For example:

```
JUMP (PC + P3) ;

CALL (PC + P0) ;
```

Condition Code Flag

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

1. A conditional branch is resolved by the value in CC.
2. A Data register value may be copied into CC, and the value in CC may be copied to a Data register.
3. The BITTST instruction accesses the CC flag.
4. A status flag may be copied into CC, and the value in CC may be copied to a status flag.
5. CC may be set to the result of a Pointer register comparison.
6. CC may be set to the result of a Data register comparison.

7. Some shifter instructions (rotate or BXOR) use CC as a portion of the shift operand/result.
8. Test and set instructions can set and clear the CC bit.

These eight ways of accessing the CC bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic flags. A single bit resides in the instruction encoding that specifies the interpretation for the value of CC. The interpretation is to "branch on true" or "branch on false."

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

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

Conditional Branches

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

For example, the following instruction tests the CC flag and, if it is positive, jumps to a location identified by the label dest_address:

```
IF CC JUMP dest_address ;
```

Conditional Register Move

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

Example code:

```
IF CC R0 = P0 ;
```

Branch Prediction

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

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

The branch latency for conditional branches is as follows:

- If prediction was “not to take branch,” and branch was actually not taken: 0 CCLK cycles.
- If prediction was “not to take branch,” and branch was actually taken: 8 CCLK cycles.
- If prediction was “to take branch,” and branch was actually taken: 4 CCLK cycles.
- If prediction was “to take branch,” and branch was actually not taken: 8 CCLK cycles.

For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the Instruction Fetch address bus at the beginning of the EX1 stage. All unconditional branches have a latency of 4 CCLK cycles.

Consider the example in Table 4-4.

Table 4-4. Branch Prediction

Instruction	Description
`If CC JUMP dest (bp)`	This instruction tests the CC flag, and if it is set, jumps to a location, identified by the label, dest. If the CC flag is set, then the branch is correctly predicted and the branch latency is reduced. Otherwise the branch is incorrectly predicted and the branch latency increases.

Loops and Sequencing

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

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

A loopback only occurs when the count is greater than or equal to 2. If the count is non-zero, then the count is decremented by 1. For example, consider the case of a loop with two iterations. At the beginning, the count is 2. Upon reaching the first loop end, the count is decremented to 1 and the program flow jumps back to the top of the loop (to execute a second time). Upon reaching the end of the loop again, the count is decremented to zero but no loopback occurs (because the body of the loop has already been executed twice).

Since there are two loop units, loop unit 1 is assigned higher priority so that it can be used as the inner loop in a nested loop structure. In other words, a loopback caused by loop unit 1 on a particular instruction (PC==LB1, LC1>=2) will prevent loop unit 0 from looping back on that same instruction, even if the address matches. Loop unit 0 is allowed to loop back only after the loop count 1 is exhausted.

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

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

```
P5 = 0x20 ;
LSETUP ( lp_start, lp_end ) LC0 = P5 ;
lp_start :
R5 = R0 + R1 ( ns ) || R2 = [P2++] || R3 = [ I1++] ;
lp_end : R5 = R5 + R2 ;
```

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

- LC[1:0] – the Loop Count registers
- LT[1:0] – the Loop Top address registers
- LB[1:0] – the Loop Bottom address registers

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

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

When LCx = 0, the loop is disabled, and a single pass of the code executes.

Table 4-5. Loop Registers

First/Last Address of the Loop	PC-Relative Offset Used to Compute the Loop Start Address	Effective Range of the Loop Start Instruction
Top / First	5-bit signed immediate; must be a multiple of 2.	0 to 30 bytes away from LSETUP instruction.
Bottom / Last	11-bit signed immediate; must be a multiple of 2.	0 to 2046 bytes away from LSETUP instruction (the defined loop can be 2046 bytes long).

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

A four-cycle latency occurs on the first loopback when the LSETUP specifies a non-zero start offset (lp_start). Therefore, zero start offsets are preferred.

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

Events and Sequencing

The Event Controller of the processor manages five types of activities:

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

- Exceptions
- Interrupts

Note the word `event` describes all five types. The Event Controller manages fifteen events in all: Emulation, Reset, NMI, Exception, and eleven interrupts.

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

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

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

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

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

Table 4-6. System and Core Event Mapping

	Event Source	Core Event Name
core events	emulation (highest priority)	EMU
	reset	RST
	NMI	NMI
	exception	EVX
	reserved	–
	hardware error	IVHW
	core timer	IVTMR
system interrupts	PLL wakeup interrupt DMA controller 0 error (generic) DMA controller 1 error (generic) PPI error interrupt SPORT0 error interrupt SPORT1 error interrupt SPORT2 error interrupt SPORT3 error interrupt MXVR synchronous data interrupt SPI0 error interrupt SPI1 error interrupt SPI2 error interrupt UART0 error interrupt UART1 error interrupt UART2 error interrupt CAN error interrupt	IVG7
	real-time clock interrupts DMA0 interrupt (PPI)	IVG8

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

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

System Interrupt Processing

Referring to Figure 4-5 on page 4-23, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISRx logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine has not yet cleared the interrupt).
2. SIC_IWRx checks to see if it should wake up the core from an idled state or Sleep mode based on this interrupt request.
3. SIC_IMASKx masks off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IARx registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7-IVG15), determine the core priority of interrupt A.
5. ILAT adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's interrupt service routine (ISR).

When the event vector for interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

When the interrupt service routine for interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISRx bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

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

If multiple interrupt sources share a single core interrupt, then the ISR must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

System Peripheral Interrupts

The processor system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:

- Peripheral interrupt source
- Peripheral interrupt ID used in the system interrupt assignment registers (SIC_IARx). See "System Interrupt Assignment (SIC_IARx) Registers" on page 4-33.
- General-purpose interrupt of the core to which the interrupt maps at reset
- The core interrupt ID used in the system interrupt assignment registers (SIC_IARx). See "System Interrupt Assignment (SIC_IARx) Registers" on page 4-33.

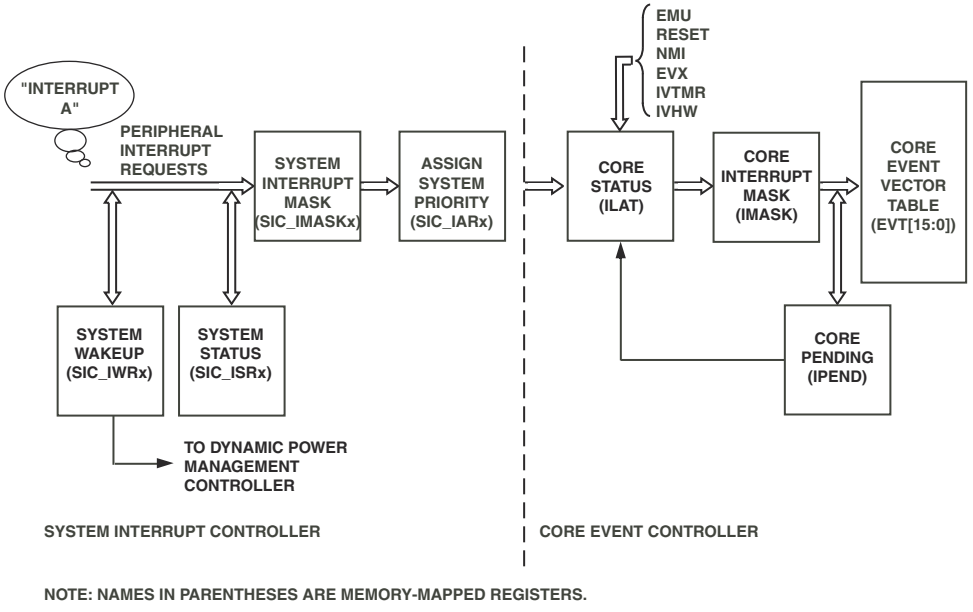


Figure 4-5. Interrupt Processing Block Diagram

Table 4-7. Peripheral Interrupt Source Reset State

Peripheral interrupt Source	Peripheral interrupt ID	General-purpose interrupt (Assignment at Reset)	Core interrupt ID
PLL wakeup interrupt	0	IVG7	0
DMA controller 0 error (generic)	1	IVG7	0
PPI error interrupt	2	IVG7	0
SPORT0 error interrupt	3	IVG7	0
SPORT1 error interrupt	4	IVG7	0
SPI0 error interrupt	5	IVG7	0
UART0 error interrupt	6	IVG7	0

Table 4-7. Peripheral Interrupt Source Reset State (Cont'd)

Peripheral interrupt Source	Peripheral interrupt ID	General-purpose interrupt (Assignment at Reset)	Core interrupt ID
Real-time Clock interrupts (alarm, second, minute, hour, countdown)	7	IVG8	1
DMA0 interrupt (PPI)	8	IVG8	1
DMA1 interrupt (SPORT0 receive)	9	IVG9	2
DMA2 interrupt (SPORT0 transmit)	10	IVG9	2
DMA3 interrupt (SPORT1 receive)	11	IVG9	2
DMA4 interrupt (SPORT1 transmit)	12	IVG9	2
DMA5 interrupt (SPI0)	13	IVG10	3
DMA6 interrupt (UART0 receive)	14	IVG10	3
DMA7 interrupt (UART0 transmit)	15	IVG10	3
Timer0 interrupt	16	IVG11	4
Timer1 interrupt	17	IVG11	4
Timer2 interrupt	18	IVG11	4
Programmable flag interrupt A	19	IVG12	5
Programmable flag interrupt B	20	IVG12	5
DMA8/9 interrupt (memory DMA stream 0)	21	IVG13	6
DMA10/11 interrupt (memory DMA stream 1)	22	IVG13	6
Software watchdog timer interrupt	23	IVG13	6
DMA controller 1 error (generic)	24	IVG7	0
SPORT2 error interrupt	25	IVG7	0
SPORT3 error interrupt	26	IVG7	0
MXVR synchronous data interrupt	27	IVG7	0
SPI1 error interrupt	28	IVG7	0
SPI2 error interrupt	29	IVG7	0

Table 4-7. Peripheral Interrupt Source Reset State (Cont'd)

Peripheral interrupt Source	Peripheral interrupt ID	General-purpose interrupt (Assignment at Reset)	Core interrupt ID
UART1 error interrupt	30	IVG7	0
UART2 error interrupt	31	IVG7	0
CAN error interrupt	32	IVG7	0
DMA8 interrupt (SPORT2 receive)	33	IVG9	2
DMA9 interrupt (SPORT2 transmit)	34	IVG9	2
DMA10 interrupt (SPORT3 receive)	35	IVG9	2
DMA11 interrupt (SPORT3 transmit)	36	IVG9	2
DMA12 interrupt	37	IVG9	2
DMA13 interrupt	38	IVG9	2
DMA14 interrupt (SPI1)	39	IVG10	3
DMA15 interrupt (SPI2)	40	IVG10	3
DMA16 interrupt (UART1 RX)	41	IVG10	3
DMA17 interrupt (UART1 TX)	42	IVG10	3
DMA18 interrupt (UART2 RX)	43	IVG10	3
DMA19 interrupt (UART2 TX)	44	IVG10	3
TWI0 interrupt	45	IVG11	4
TWI1 interrupt	46	IVG11	4
CAN receive interrupt	47	IVG11	4
CAN transmit interrupt	48	IVG11	4
MDMA1 stream 0 (memory DMA)	49	IVG13	6
MDMA1 stream 1 (memory DMA)	50	IVG13	6
MXVR status interrupt	51	IVG11	4
MXVR control message interrupt	52	IVG11	4
MXVR asynchronous packet interrupt	53	IVG11	4
Reserved	54-63	-	-

The peripheral interrupt structure of the processor is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in Table 4-7.

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

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

System Interrupt Wakeup-Enable (SIC_IWRx) Registers

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

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

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

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

Note the wake-up function is independent of the interrupt mask function. If an interrupt source is enabled in SIC_IWRx but masked off in SIC_IMASKx, the core wakes up from the idle or Sleep modes, but it does not vector to an interrupt service routine.

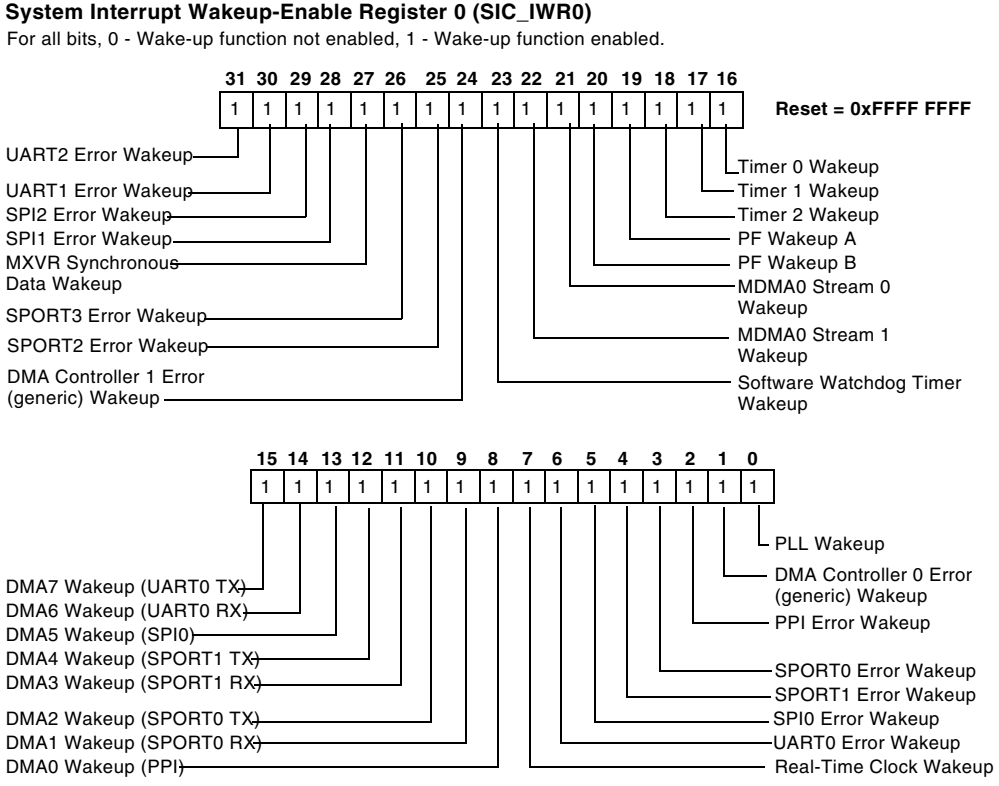


Figure 4-6. System Interrupt Wakeup-Enable Register 0 (SIC_IWR0)

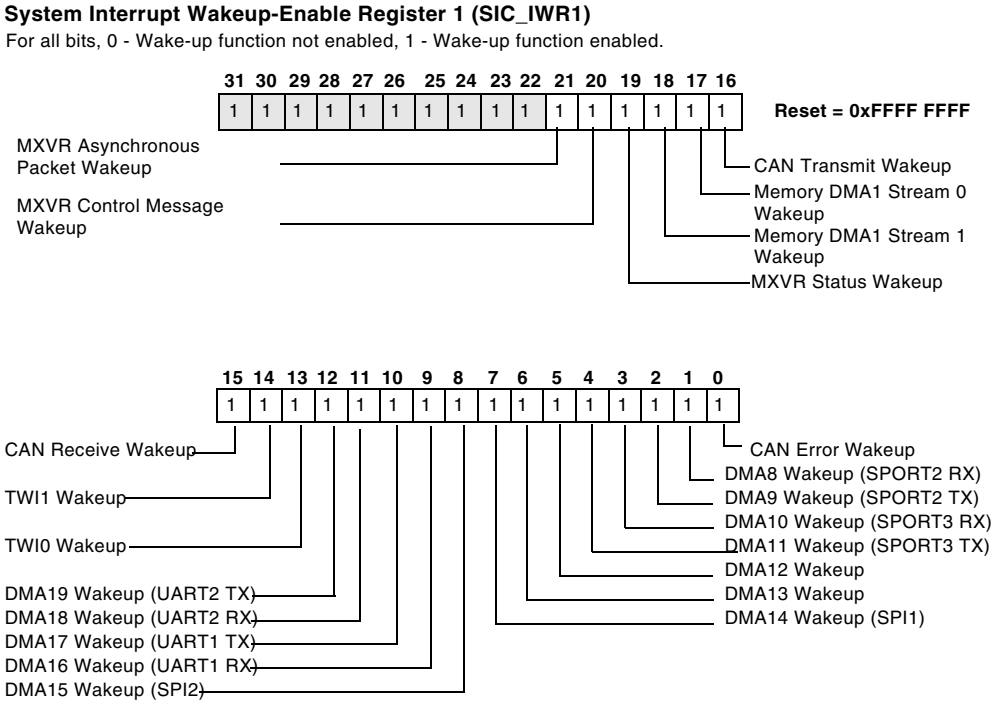


Figure 4-7. System Interrupt Wakeup-Enable Register 1 (SIC_IWR1)

System Interrupt Status (SIC_ISRx) Registers

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

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

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

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

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

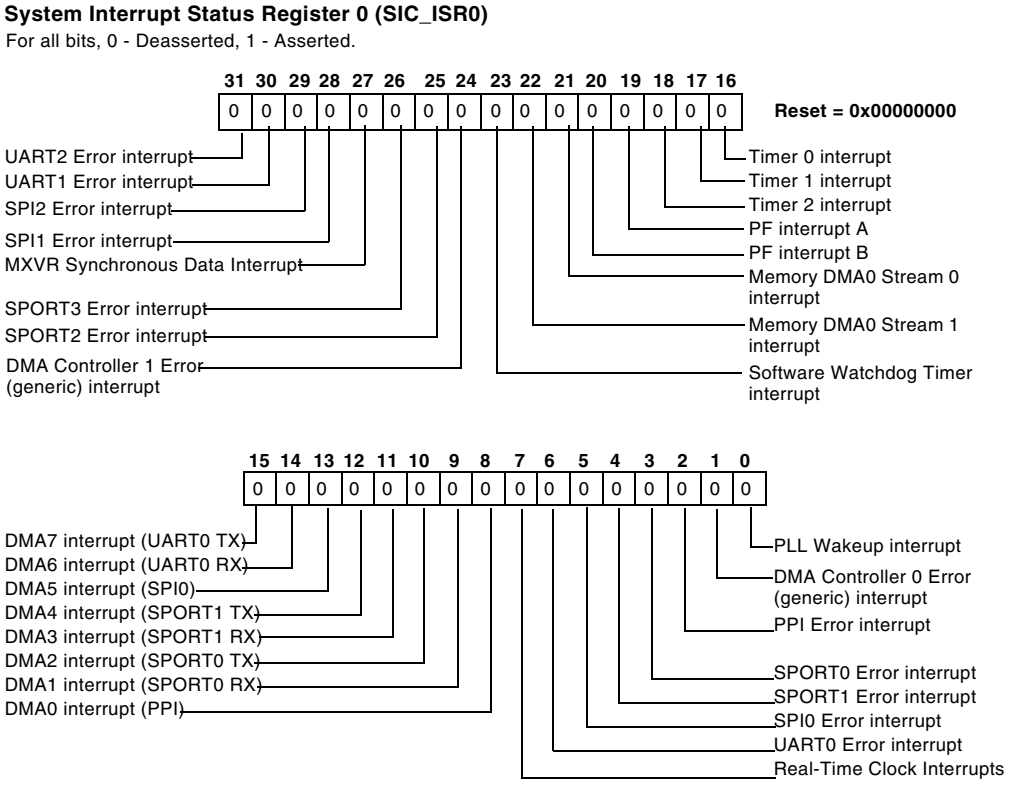


Figure 4-8. System Interrupt Status Register 0 (SIC_ISR0)

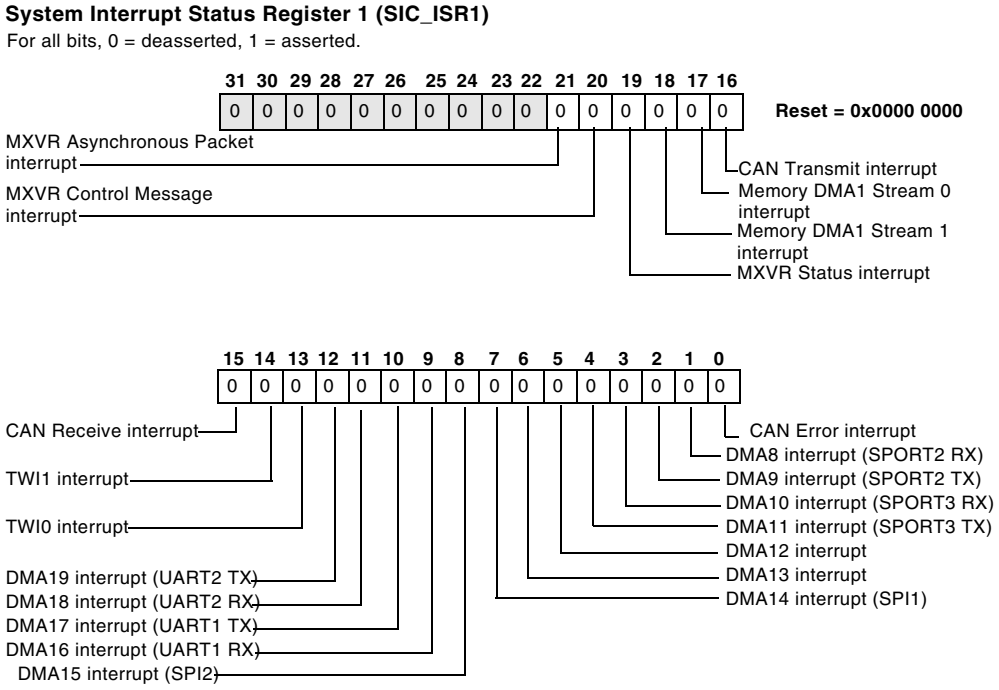


Figure 4-9. System Interrupt Status Register 1 (SIC_ISR1)

System Interrupt Mask (SIC_IMASKx) Registers

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

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

Although this register can be read from or written to at any time (in Supervisor mode), it should be configured in the reset initialization sequence before enabling interrupts.

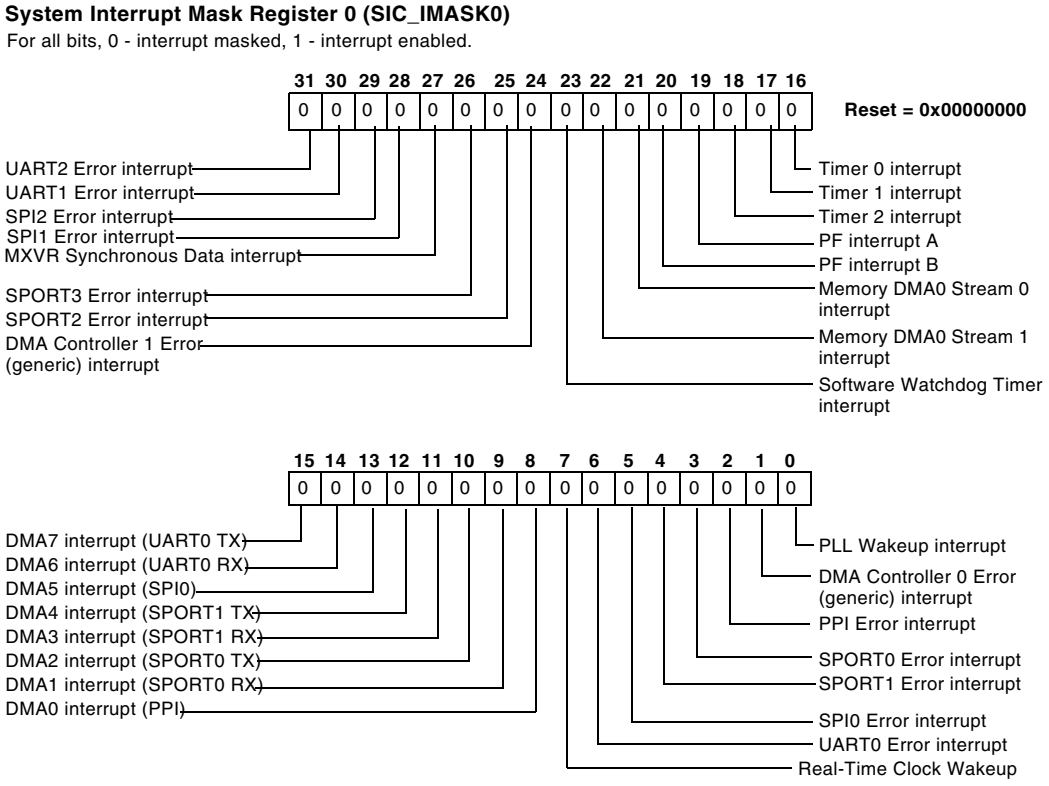


Figure 4-10. System Interrupt Mask Register 0 (SIC_IMASK0)

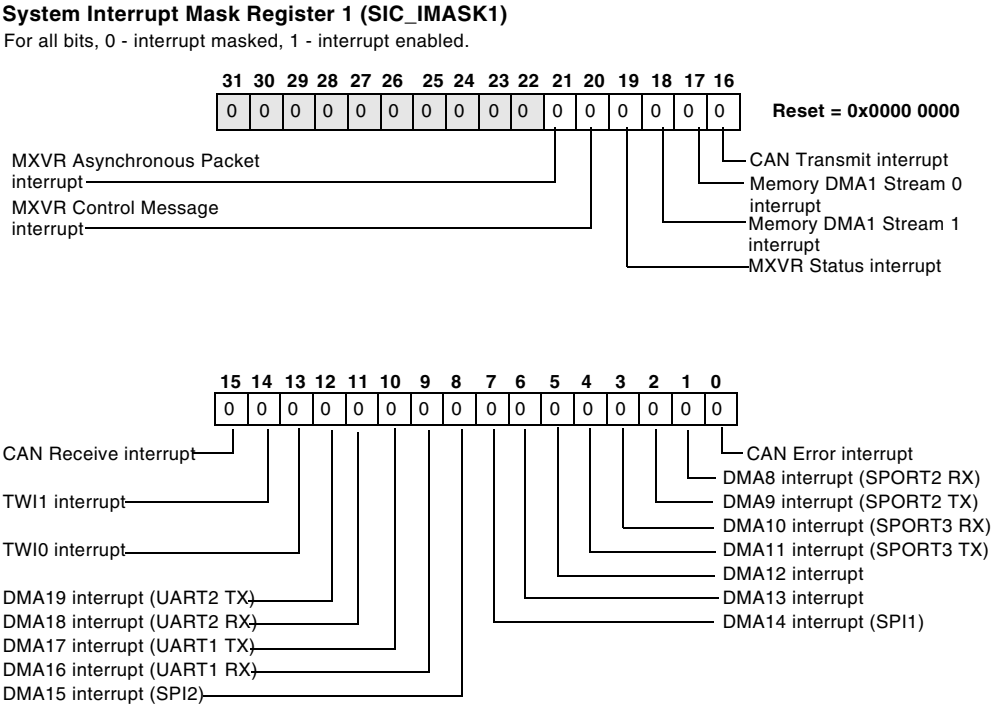


Figure 4-11. System Interrupt Mask Register 1 (SIC_IMASK1)

System Interrupt Assignment (SIC_IARx) Registers

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

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

System Interrupt Assignment Register 0 (SIC_IAR0)

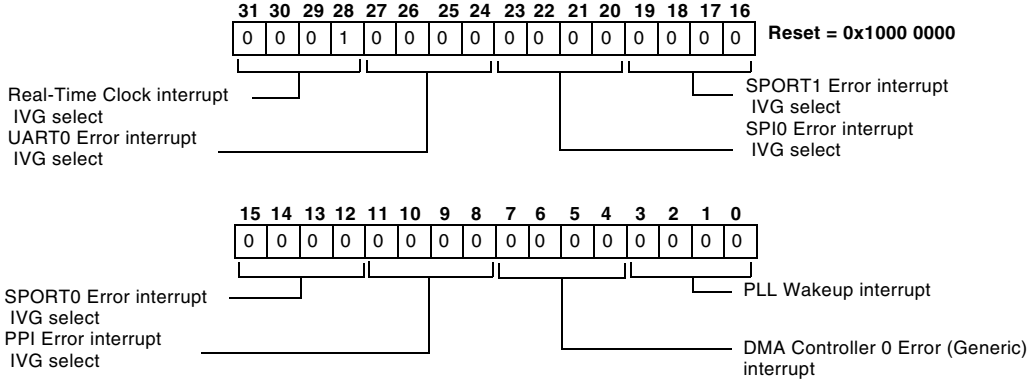


Figure 4-12. System Interrupt Assignment Register 0 (SIC_IAR0)

System Interrupt Assignment Register 1 (SIC_IAR1)

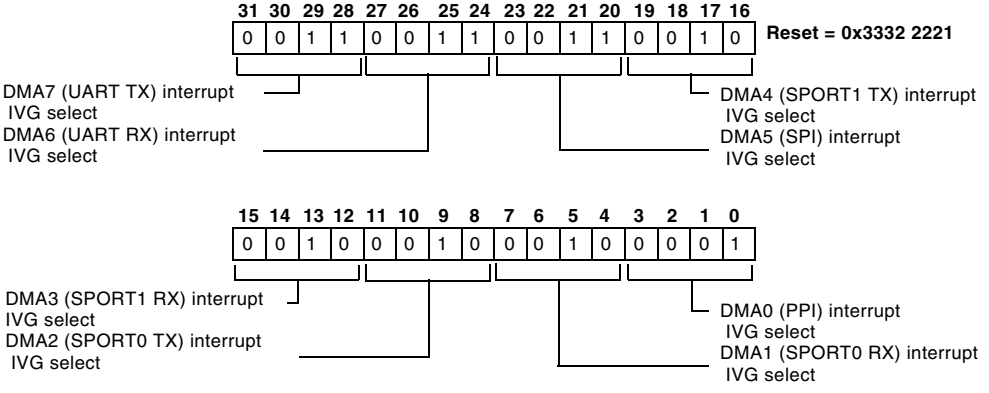


Figure 4-13. System Interrupt Assignment Register 1 (SIC_IAR1)

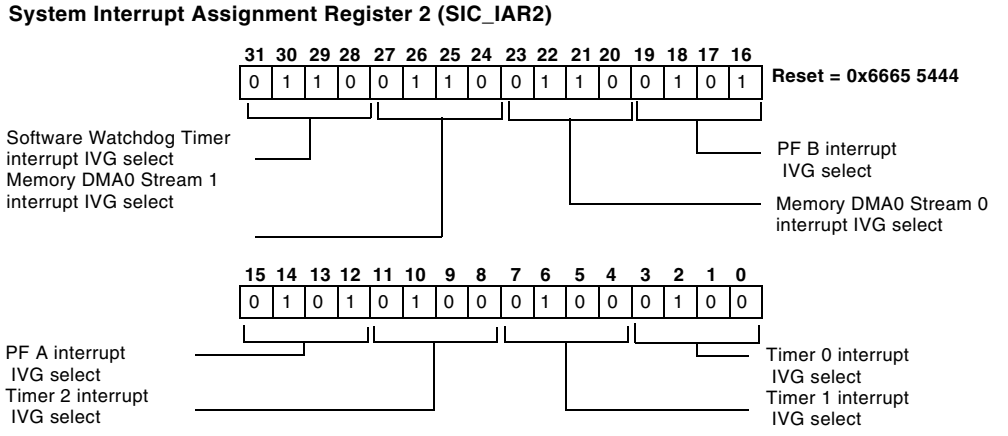


Figure 4-14. System Interrupt Assignment Register 2 (SIC_IAR2)

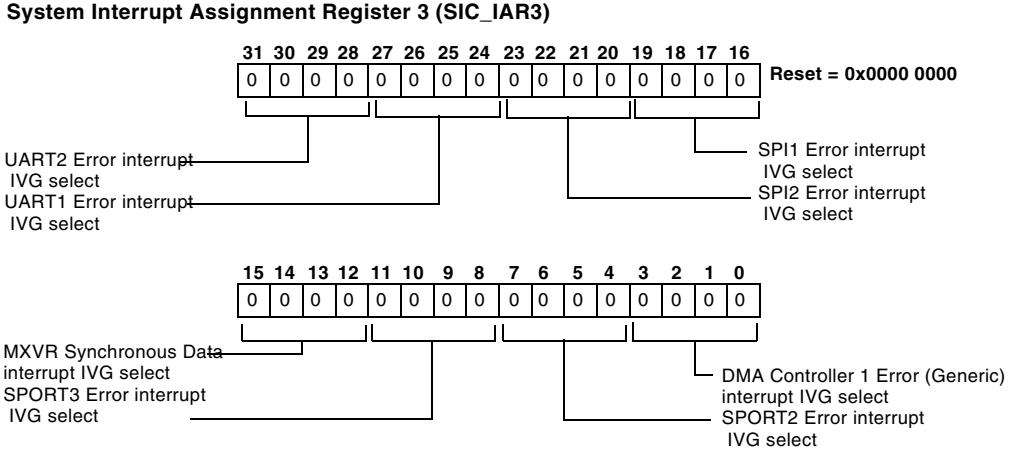


Figure 4-15. System Interrupt Assignment Register 3 (SIC_IAR3)

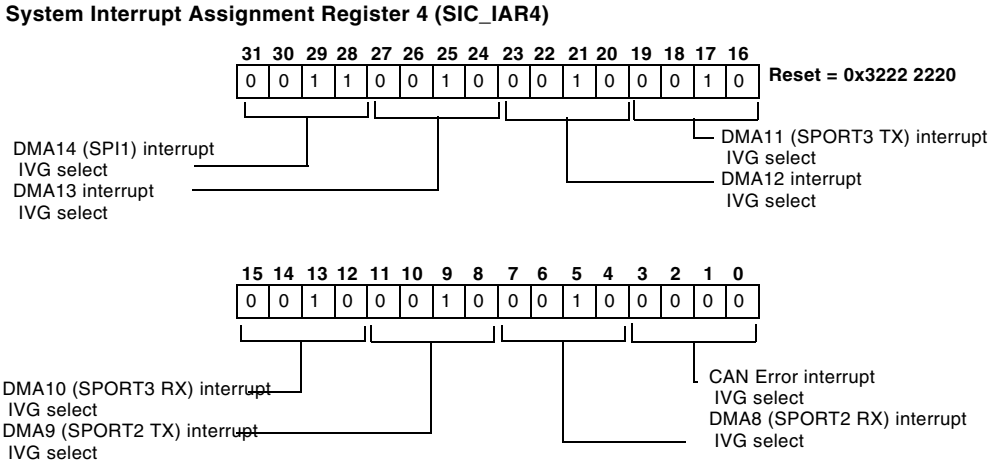


Figure 4-16. System Interrupt Assignment Register 4 (SIC_IAR4)

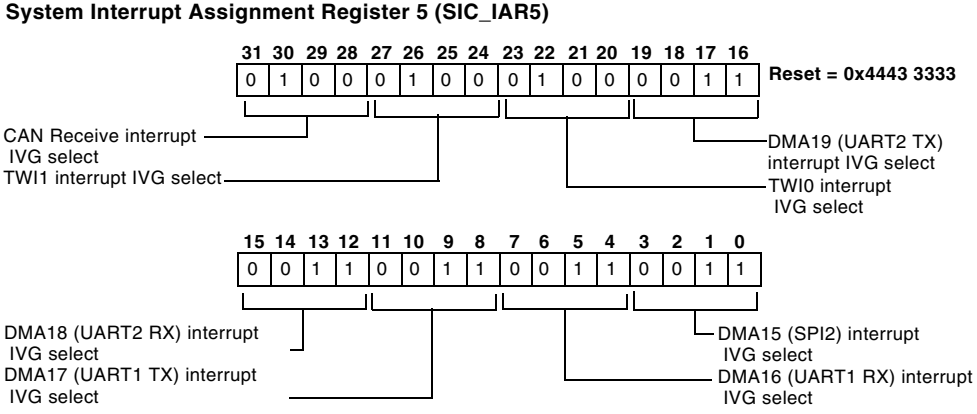


Figure 4-17. System Interrupt Assignment Register 5 (SIC_IAR5)

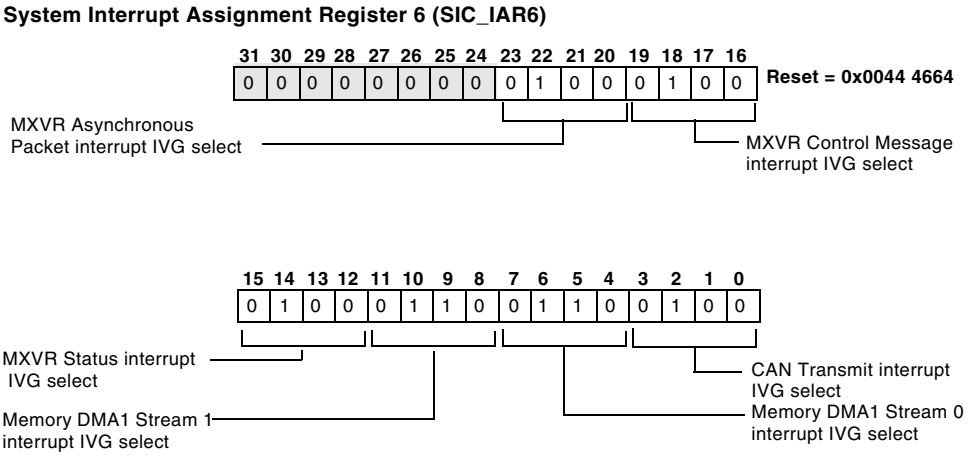


Figure 4-18. System Interrupt Assignment Register 6 (SIC_IAR6)

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

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

Table 4-8. IVG-Select Definitions

General-purpose interrupt	Value in SIC_IARx
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5
IVG13	6
IVG14	7
IVG15	8

Core Event Controller Registers

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

`IMASK` - interrupt mask

`ILAT` - interrupt latch

`IPEND` - interrupts pending

These three registers are accessible in Supervisor mode only.

Core Interrupt Mask (IMASK) Register

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

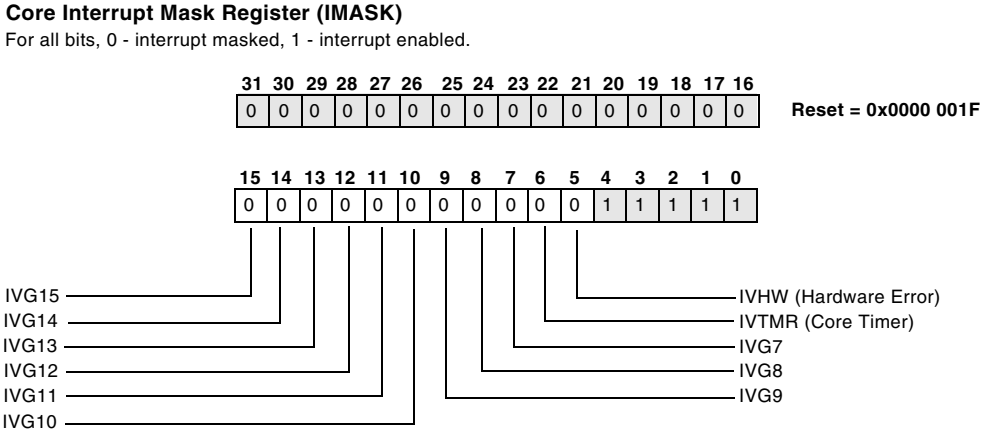


Figure 4-19. Core Interrupt Mask Register

Core Interrupt Latch (ILAT) Register

Each bit in ILAT indicates that the corresponding event is latched, but not yet accepted into the processor (see Figure 4-20 on page 4-40). The bit is reset before the first instruction in the corresponding ISR is executed. At the point the interrupt is accepted, ILAT[N] is cleared and IPEND[N] is set simultaneously. The ILAT register can be read in supervisor mode. Writes to ILAT are used to clear bits only (in Supervisor mode). To clear bit N from ILAT, first make sure that IMASK[N]==0, and then write ILAT[N]=1. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of servicing them.

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

Only the JTAG $\overline{\text{TRST}}$ pin can clear ILAT[0].

Core Interrupt Latch Register (ILAT)

Reset value for bit 0 is emulator-dependent. For all bits, 0 - interrupt not latched, 1 - interrupt latched.

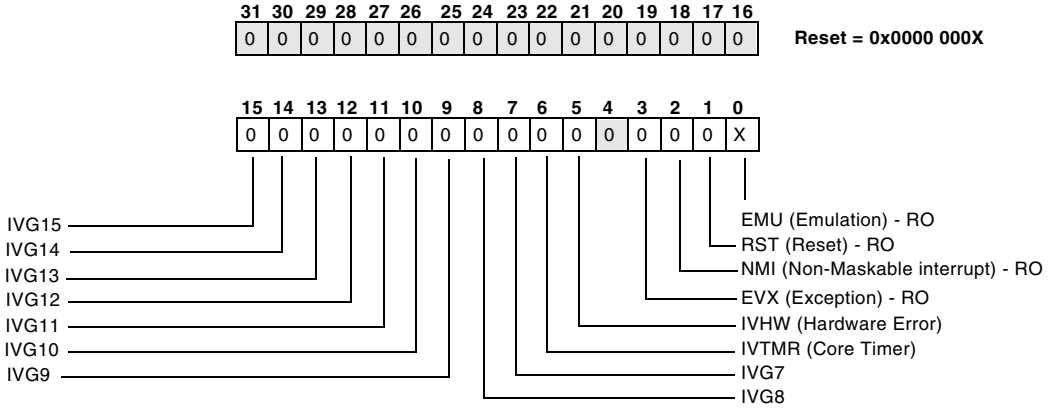


Figure 4-20. Core Interrupt Latch Register

Core Interrupts Pending (IPEND) Register

The IPEND register keeps track of all currently nested interrupts (see Figure 4-23 on page 4-58). Each bit in IPEND indicates that the corresponding interrupt is currently active or nested at some level. It may be read in Supervisor mode, but not written. The IPEND[4] bit is used by the Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

When an event is processed, the corresponding bit in IPEND is set. The least significant bit in IPEND that is currently set indicates the interrupt that is currently being serviced. At any given time, IPEND holds the current status of all nested events.

Core Interrupt Pending Register (IPEND)
RO. For all bits except bit 4, 0 - No interrupt pending, 1 - interrupt pending or active.

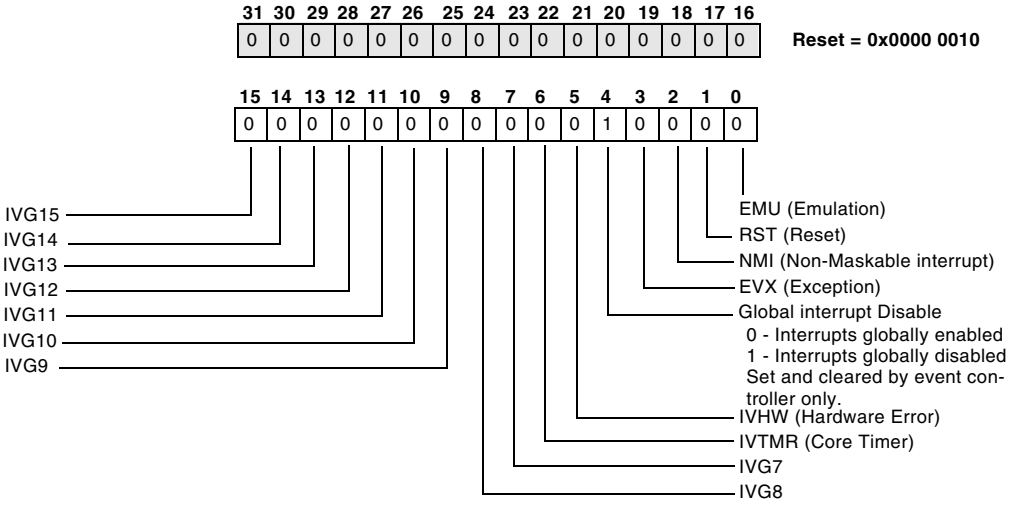


Figure 4-21. Core Interrupt Pending Register

Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the `CLI Dreg` instruction and re-enabled with the `STI Dreg` instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears `IMASK[15:5]` after saving `IMASK`'s current state. See "Enable Interrupts" and "Disable Interrupts" in the "External Event Management" chapter in *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

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

Event Vector Table

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

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

The following table lists events by priority. Each event has a corresponding bit in the event state registers `ILAT`, `IMASK`, and `IPEND`.

Table 4-9. Core Event Vector Table

Event Number	Event Class	Name	MMR Location	Notes
EVT0	Emulation	EMU	0xFFE0 2000	Highest priority. Vector address is provided by JTAG.
EVT1	Reset	RST	0xFFE0 2004	
EVT2	NMI	NMI	0xFFE0 2008	
EVT3	Exception	EVX	0xFFE0 200C	
EVT4	Reserved	Reserved	0xFFE0 2010	Reserved vector.
EVT5	Hardware Error	IVHW	0xFFE0 2014	
EVT6	Core Timer	IVTMR	0xFFE0 2018	
EVT7	Interrupt 7	IVG7	0xFFE0 201C	
EVT8	Interrupt 8	IVG8	0xFFE0 2020	
EVT9	Interrupt 9	IVG9	0xFFE0 2024	
EVT10	Interrupt 10	IVG10	0xFFE0 2028	
EVT11	Interrupt 11	IVG11	0xFFE0 202C	
EVT12	Interrupt 12	IVG12	0xFFE0 2030	
EVT13	Interrupt 13	IVG13	0xFFE0 2034	
EVT14	Interrupt 14	IVG14	0xFFE0 2038	
EVT15	Interrupt 15	IVG15	0xFFE0 203C	Lowest priority.

Emulation

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

For detailed information about emulation, see Chapter 23, “Blackfin Processor Debug”.

Reset

The reset interrupt (RST) can be initiated via the $\overline{\text{RESET}}$ pin or through expiration of the Watchdog timer. This location differs from that of other interrupts in that its content is read-only. Writes to this address change the register but do not change where the processor vectors upon reset. The processor always vectors to the reset vector address upon reset. For more information, see "Reset State" on page 3-11 and "Booting Methods" on page 3-18.

The core has an output that indicates that a double-fault has occurred. This is a non-recoverable state. The system (via the SWRST register) can be programmed to send a reset request if a double-fault condition is detected. Subsequently, the reset request forces a system reset for core and peripherals.

The reset vector is determined by the processor system. It points to the start of the on-chip boot ROM, or to the start of external asynchronous memory, depending on the state of the BMODE[1:0] pins. Refer to Table 2-10 on page 2-44.

Table 4-10. Reset Vector Addresses

Boot Source	BMODE[1:0]	Execution Start Address
Execute from 16-bit external memory	b#00	0x2000 0000
Boot from 8-bit or 16-bit external flash memory (or from on-chip flash memory, for ADSP-BF539F processors)	b#01	0xEF00 0000
Boot from an SPI host in SPI Slave Mode	b#10	0xEF00 0000
Boot from an 8-/16-/24-bit addressable SPI in SPI Master Mode	b#11	0xEF00 0000

If the BMODE[1:0] pins indicate either booting from flash or serial ROM, the reset vector points to the start of the internal boot ROM, where a small bootstrap kernel resides. The bootstrap code reads the System Reset Configuration register (SYSCR) to determine the value of the BMODE[1:0] pins, which determine the appropriate boot sequence. For information about the boot ROM, see "Booting Methods" on page 3-18.

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

NMI (Non-Maskable Interrupt)

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

Any exception in any event handler of exception level or above (including NMI) will trigger a "double-fault" condition, and the address of the excepting instruction will be written to RETX.

The polarity of the $\overline{\text{NMI}}$ input signal has changed from previous Blackfin products. The ADSP-BF539/ADSP-BF539F input is active low.

Exceptions

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

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

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

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

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

Usually the RETX register contains the correct address to return to. To skip over an excepting instruction, take care in case the “next” address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case the proper “next” address would be the loop top.

The EXCAUSE[5:0] field in the Sequencer status register (SEQSTAT) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to Table 4-11 for a list of events that cause exceptions.

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

For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

Table 4-11. Events That Cause Exceptions

Exception	EXCAUSE [5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Force Exception instruction EXCPT with 4-bit field m	m-field	S	Instruction provides four bits of EXCAUSE.
Single step	0x10	S	When the processor is in single-step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by an emulation trace buffer overflow	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the trace unit control register).
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	Attempted read or write to supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for supervisor use: supervisor only registers, all MMRs, and supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the memory management unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access misaligned address violation	0x24	E	Attempted misaligned data memory or data cache access.

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	EXCAUSE [5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the watchpoint instruction address control register (WPIACTL) is set.

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	EXCAUSE [5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a supervisor register or instruction from user mode. Supervisor resources are registers and instructions that are reserved for supervisor use: supervisor only registers, all MMRs, and supervisor only instructions.

If an instruction causes multiple exceptions, only the exception with the highest priority is taken. Table 4-12 ranks exceptions by descending priority.

Table 4-12. Exceptions by Descending Priority

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal use protected resource	0x2E
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT instruction	m- field
20	Single Step	0x10
21	Trace Buffer	0x11

Exceptions While Executing an Exception Handler

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

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

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

Hardware-Error Interrupt

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

The hardware-error interrupt is generated by:

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

The list of supported hardware conditions, with their related `HWERRCAUSE` codes, appears in Table 4-13. The bit code for the most recent error appears in the `HWERRCAUSE` field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The hardware error interrupt remains active as long as any of the error conditions remain active.

Table 4-13. Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Binary)	HWERRCAUSE (Hexadecimal)	Notes/Examples
System MMR Error	0b00010	0x02	An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.
External Memory Addressing Error	0b00011	0x03	An access to reserved or uninitialized memory was attempted.
Performance Monitor Overflow	0b10010	0x12	Refer to "Performance Monitoring Unit" on page 23-20.
RAISE 5 instruction	0b11000	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other bit combinations.	All other values.	

Core Timer

The core timer interrupt (`IVTMR`) is triggered when the core timer value reaches zero. See Chapter 16, "Timers".

General-Purpose Interrupts (IVG7-IVG15)

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

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

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

Servicing Interrupts

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

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

Servicing the highest priority interrupt involves the following actions:

1. The interrupt vector in the EVT becomes the next fetch address.
2. On an interrupt, most instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.
3. The return address is saved in the appropriate return register.

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

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

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

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

Bit IPEND[4] is also set to disable all interrupts until the return address in RETI is saved.

Nesting of Interrupts

Interrupts are handled either with or without nesting.

Non-Nested Interrupts

If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in RETI. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the RTI instruction must be executed, because the return address is already held in the RETI register.

Figure 4-22 shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

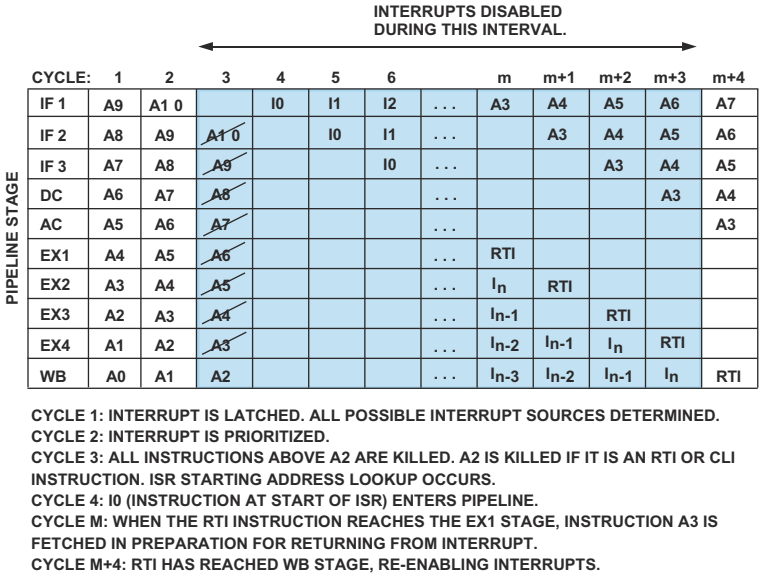


Figure 4-22. Non-Nested Interrupt Handling

Nested Interrupts

If nested interrupts are desired, the return address to the interrupted point in the original interrupt service routine (ISR) must be explicitly saved and subsequently restored when execution of the nested ISR has completed. Nesting is enabled by pushing the return address currently held in RETI to the Supervisor stack (`[--SP] = RETI`), which is typically done early in the ISR prolog of the lower priority interrupt. This clears the global interrupt disable bit `IPEND[4]`, enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push `RETI` (`[--SP]=RETI`) and pop `RETI` (`RETI=[SP++]`) use the Supervisor stack.

Figure 4-23 illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an ISR, resulting in only a short duration where interrupts are globally disabled.

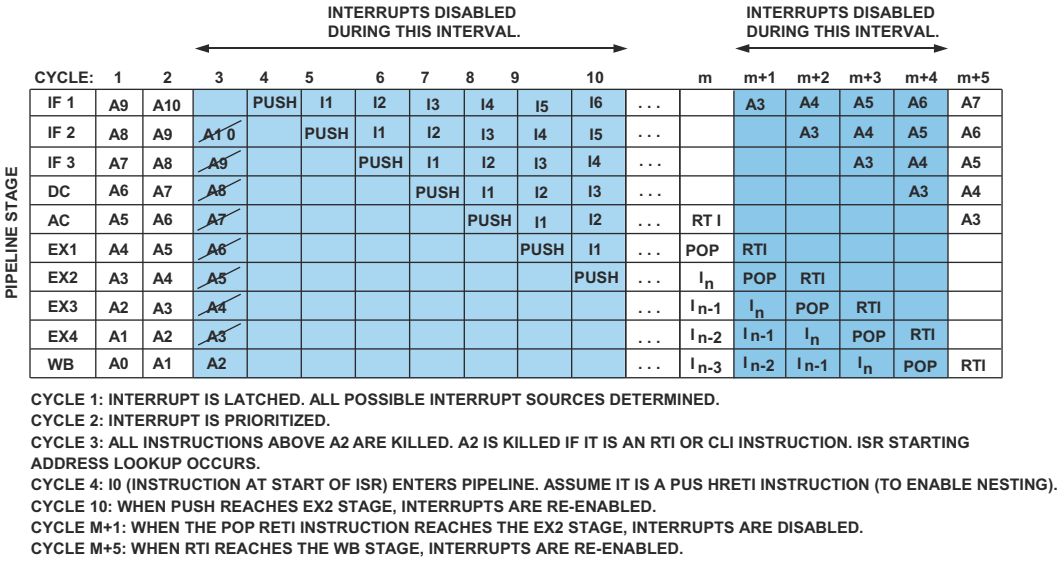


Figure 4-23. Nested Interrupt Handling

Example Prolog Code for Nested Interrupt Service Routine

```
/* Prolog code for nested interrupt service routine. Push return
address in RETI into Supervisor stack, ensuring that interrupts
are back on. Until now, interrupts have been suspended. */
ISR :
[--SP] = RETI ; /* Enables interrupts and saves return address to
stack. */
[--SP] = ASTAT ;
[--SP] = FP ;
[-- SP] = (R7:0, P5:0) ;
/* Body of service routine. Note none of the processor resources
(accumulators, DAGs, loop counters and bounds) have been saved.
It's assumed that this interrupt service routine does not use
them. */
```

Example Epilog Code for Nested Interrupt Service Routine

```
/* Epilog code for nested-interrupt service routine. Restore
ASTAT, Data and Pointer registers. Popping RETI from Supervisor
stack ensures that interrupts are suspended between load of
return address and RTI. */
( R7:0, P5:0 ) = [SP++] ;
FP   = [SP++] ;
ASTAT = [SP++] ;
RETI = [SP++] ;
/* Execute RTI, which jumps to return address, re-enables inter-
rupts, and switches to User mode if this is the last nested
interrupt in service. */
RTI ;
```

The `RTI` instruction causes the return from an interrupt. The return address is popped into the `RETI` register from the stack, an action that suspends interrupts from the time that `RETI` is restored until `RTI` finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the `RETI` register.

Next, the `RTI` instruction clears the highest priority bit that is currently set in `IPEND`. The processor then jumps to the address pointed to by the value in the `RETI` register and re-enables the interrupts by clearing `IPEND[4]`.

Logging of Nested Interrupt Requests

The SIC detects level-sensitive interrupt requests from the peripherals. The CEC provides edge-sensitive detection for its general-purpose interrupts (`IVG7-IVG15`). Consequently, the SIC generates a synchronous interrupt pulse to the CEC and then wait for interrupt acknowledgement from the CEC. When the interrupt has been acknowledged by the core (via assertion of the appropriate `IPEND` output), the SIC generates another

synchronous interrupt pulse to the CEC if the peripheral interrupt is still asserted. This way, the system does not lose peripheral interrupt requests that occur during servicing of another interrupt.

Because multiple interrupt sources can map to a single core processor general-purpose interrupt, multiple pulse assertions from the SIC can occur simultaneously, before, or during interrupt processing for an interrupt event that is already detected on this interrupt input. For a shared interrupt, the IPEND interrupt acknowledge mechanism described above re-enables all shared interrupts. If any of the shared interrupt sources are still asserted, at least one pulse is again generated by the SIC. The interrupt register registers indicate the current state of the shared interrupt sources.

Exception Handling

Interrupts and exceptions treat instructions in the pipeline differently:

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the push and pop instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (RTX, RTN, and RTE) do clear the least significant bit currently set in IPEND.

Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force interrupt/reset instruction (`RAISE`).

When deferring the processing of an exception to lower priority interrupt `IVGx`, the system must guarantee that `IVGx` is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than `IVGx` occurs, it is acceptable to enter the high priority interrupt before `IVGx`.

Example Code for an Exception Handler

The following code is for an exception routine handler with deferred processing:

Listing 4-1. Exception Routine Handler with Deferred Processing

```
/* Determine exception cause by examining EXCAUSE field in SEQ-
STAT (first save contents of R0, P0, P1 and ASTAT in Supervisor
SP) */
[--SP] = R0 ;
[--SP] = P0 ;
[--SP] = P1 ;
[--SP] = ASTAT ;
R0 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R0 << = 26 ;
```

```
R0 >> = 26 ;
/* Using jump table EVTABLE, jump to the event pointed by R0 */
P0 = R0 ;
P1 = _EVTABLE ;
P0 = P1 + ( P0 << 1 ) ;
R0 = W [ P0 ] (Z) ;
P1 = R0 ;
JUMP ( PC + P1 ) ;
/* The entry point for an event is as follows. Here, processing
is deferred to low-priority interrupt IVG15. Also, parame-
ter-passing would typically be done here. */
_EVENT1 :
RAISE 15 ;
JUMP.S_EXIT ;
/* Entry for event at IVG14 */
_EVENT2 :
RAISE 14 ;
JUMP.S _EXIT ;
/* comments for other events. At the end of handler, restore R0,
P0, P1 and ASTAT, and return. */
_EXIT :
ASTAT = [SP++] ;
P1 = [SP++] ;
P0 = [SP++] ;
R0 = [SP++] ;
RTI ;
_EVTABLE :
.byte2 addr_event1 ;
.byte2 addr_event2 ;
...
.byte2 addr_eventN ;
/* The jump table EVTABLE holds 16-bit address offsets for each
event. With offsets, this code is position-independent and the
table is small.
```

```
+--------------+
| addr_event1  | _EVTABLE
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
|    . . .     |
+--------------+
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/
```

Example Code for an Exception Routine

The following code provides an example framework for an exception routine that would be jumped to from an exception handler such as that described above.

Listing 4-2. Exception Routine

```
[--SP] = RETX ;  /* Push return address on stack. No change to
ILAT or IPEND. */

/* Put body of exception routine here. */

RETX = [SP++] ;  /* To return, pop return address and jump. No
change to ILAT or IPEND. */

RTX ;  /* Return from exception. Clear IPEND[3] (Exception Pend-
ing bit). */
```

Example Code for Using Hardware Loops in an ISR

The following code provides shows the optimal method of saving and restoring when using hardware loops in an ISR:

Listing 4-3. Exception Routine

```
lhandler

<save other registers here>

[--SP] = LC0 ;  /* save loop 0 */
[--SP] = LB0 ;
[--SP] = LT0 ;

<handler code here>

/* If the handler uses loop 0, it is a good idea to have it leave
LC0 equal to zero at the end. Normally, this will happen natu-
rally as a loop is fully executed. If LC0 == 0, then LT0 and LB0
restores will not incur additional cycles. If LC0 ! = 0 when the
following pops happen, each pop will incur a 10-cycle “replay”
penalty. Popping or writing LC0 always incurs the penalty. */

LT0 = [SP++] ;
LB0 = [SP++] ;
LC0 = [SP++] ;  /* This will cause a “replay,” That is, a
10-cycle refetch. */

<restore other registers here>

RTI ;
```

Other Usability Issues

The following sections describe other usability issues.

Executing RTX, RTN, or RTE in a Lower Priority Event

Instructions RTX, RTN, and RTE are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower-priority event. To return from an interrupt, use the RTI instruction. Failure to use the correct instruction produces the following results.

If a program mistakenly uses RTX, RTN, or RTE to return from an interrupt, the core branches to the address in the corresponding return register (RETX, RETN, RETE). IPEND is modified correctly but the return address will possibly be wrong.

If a program mistakenly uses RTI or RTX to return from an NMI routine, the core branches to the address in the corresponding return register (RETI, RETX), and clears the bit in IPEND that corresponds to the return instruction.

In the case of RTX, bit IPEND[3] is cleared. In the case of RTI, the bit of the highest priority interrupt in IPEND is cleared.

Recommendation for Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the supervisor stack grows past a CPLB entry or SRAM block, it may, in fact, generate an exception.

To guarantee that the Supervisor stack never generates an exception—never overflows past a CPLB entry or SRAM block while executing the exception handler—calculate the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then allocate this amount of SRAM memory.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine (ISR) may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the ISR could be held off for many hundreds of cycles.

In order for high-priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an ISR executes from L1 memory. Figure 2-15 on page 2-65 illustrates this concept.

If an instruction load operation misses the L1 Instruction Cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the ISR in its new context, and the cache fill operation completes in the background.

Note the ISR must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation. If a load or store operation is executed in the ISR requiring one of these accesses, then the ISR is held off while the original external access is completed, before initiating the new load or store.

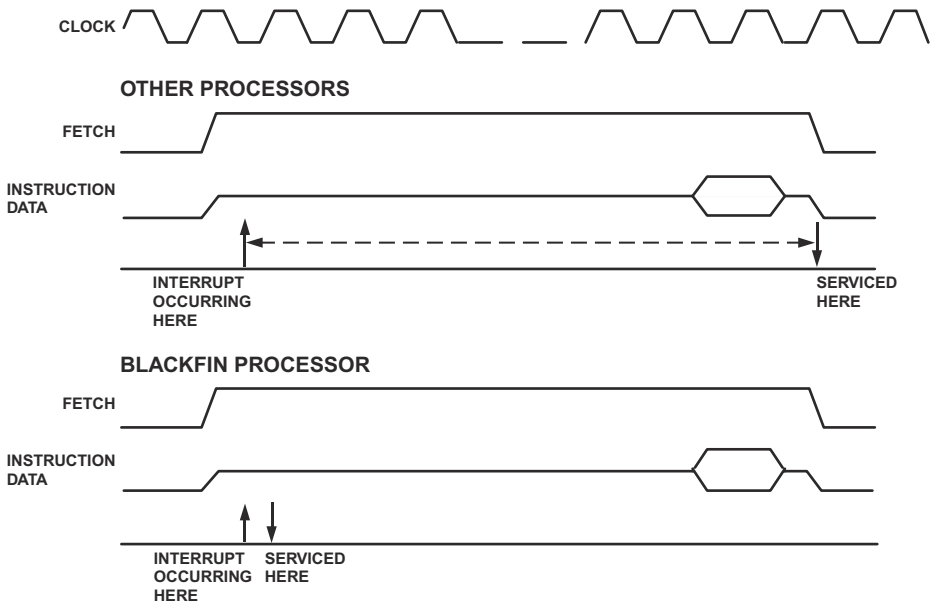


Figure 4-24. Minimizing Latency in Servicing an ISR

If the ISR finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, Chapter 6, “Memory”.

5 DATA ADDRESS GENERATORS

The Data Address Generators (DAGs) generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address.

The DAG architecture, shown in Figure 5-1 on page 5-3, supports several functions that minimize overhead in data access routines. These functions include:

- Supply address – Provides an address during a data access
- Supply address and post-modify – Provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset – Provides an address from a base with an offset without incrementing the original address pointer
- Modify address – Increments or decrements the stored address without performing a data move
- Bit-reversed carry address – Provides a bit-reversed carry address during a data move without reversing the stored address

The DAG subsystem comprises two DAG Arithmetic units, nine pointer registers, four Index registers and four complete sets of related Modify, Base, and Length registers. These registers hold the values that the DAGs use to generate addresses. The types of registers are:

- Index registers I[3:0]—unsigned 32-bit index registers hold an address pointer to memory. For example, the instruction R3 = [I0] loads the data value found at the memory location pointed to by the register I0. Index registers can be used for 16- and 32-bit memory accesses.
- Modify registers M[3:0]—signed 32-bit modify registers provide the increment or step size by which an Index register is post-modified during a register move. For example, the R0 = [I0 ++ M1] instruction directs the DAG to:

  – Output the address in register I0
  – Load the contents of the memory location pointed to by I0 into R0
  – Modify the contents of I0 by the value contained in the M1 register
- Base and length registers B[3:0] and L[3:0]—unsigned 32-bit base and length registers set up the range of addresses and the starting address of a circular buffer. Each B–L pair is always coupled with a corresponding I-register, for example I3, B3, L3. For more information on circular buffers, see "Addressing Circular Buffers" on page 5-6.
- Pointer registers P[5:0], FP, USP, and SP—32-bit pointer registers hold an address pointer to memory. The P[5:0] field, FP (frame pointer) and SP/USP (stack pointer/user stack pointer) can be manipulated and used in various instructions. For example, the instruction R3 = [P0] loads the register R3 with the 32-bit data value found at the memory location pointed to by the register P0. The pointer registers have no effect on circular buffer addressing. They can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, SP is accessible only in supervisor mode, while USP is accessible in user mode.

Do not assume the L-registers are automatically initialized to zero for linear addressing. The I-, M-, L-, and B-registers contain random values after reset. For each I-register used, programs must initialize the corresponding L-registers to zero for linear addressing or to the buffer length for circular buffer addressing.

Note all DAG registers must be initialized individually. Initializing a B-register does not automatically initialize the I-register.

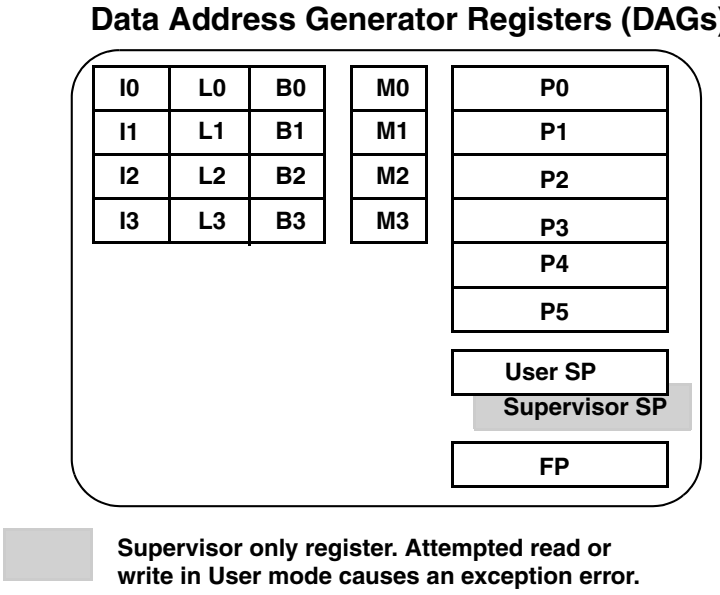


Figure 5-1. Processor DAG Registers

Addressing With DAGs

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the I-register value unchanged; then the DAG adds an M-register or immediate value to the I-register.

In indexed addressing, the DAG adds a small offset to the value in the P-register, but does not update the P-register with this new value, thus providing an offset for that particular memory access.

The processor is byte addressed. All data accesses must be aligned to the data size. In other words, a 32-bit fetch must be aligned to 32 bits, but an 8-bit store can be aligned to any byte. Depending on the type of data used, increments and decrements to the DAG registers can be by 1, 2, or 4 to match the 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

```
R0 = [ P3++ ];
```

This instruction fetches a 32-bit word, pointed to by the value in `P3`, and places it in `R0`. It then post-increments `P3` by *four*, maintaining alignment with the 32-bit access.

```
R0.L = W [ I3++ ];
```

This instruction fetches a 16-bit word, pointed to by the value in `I3`, and places it in the low half of the destination register, `R0.L`. It then post-increments `I3` by *two*, maintaining alignment with the 16-bit access.

```
R0 = B [ P3++ ] (Z) ;
```

This instruction fetches an 8-bit word, pointed to by the value in `P3`, and places it in the destination register, `R0`. It then post-increments `P3` by one, maintaining alignment with the 8-bit access. The byte value may be zero extended (as shown with the trailing Z) or sign extended into the 32-bit data register by using a trailing X.

Instructions using Index registers use an M-register or a small immediate value (+/– 2 or 4) as the modifier. Instructions using Pointer registers use a small immediate value or another P-register as the modifier. For details, see Table 5-3 on page 5-17.

Frame and Stack Pointers

In many respects, the Frame and Stack Pointer registers perform like the other P-registers, `P[5:0]`. They can act as general pointers in any of the load/store instructions, for example, `R1 = B[SP] (Z)`. However, `FP` and `SP` have additional functionality.

The Stack Pointer registers include:

- a User Stack Pointer (`USP` in Supervisor mode, `SP` in User mode)
- a Supervisor Stack Pointer (`SP` in Supervisor mode)

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias `SP`. Depending on the current processor operating mode, only one of these registers is active and accessible as `SP`:

- In User mode, any reference to `SP` (for example, stack pop `R0 = [ SP++ ] ;`) implicitly uses the `USP` as the effective address.
- In Supervisor mode, the same reference to `SP` (for example, `R0 = [ SP++ ] ;`) implicitly uses the Supervisor Stack Pointer as the effective address.

  To manipulate the User Stack Pointer for code running in Supervisor mode, use the register alias `USP`. When in Supervisor mode, a register move from `USP` (for example, `R0 = USP ;`) moves the current User Stack Pointer into `R0`. The register alias `USP` can only be used in Supervisor mode.

Some load/store instructions use FP and SP implicitly:

- FP-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions, including those for pushing and popping multiple registers
- Link/unlink instructions, which control stack frame space and manage the Frame Pointer register (FP) for that space

Addressing Circular Buffers

The DAGs support addressing circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of DAG registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The Index (I) register contains the value that the DAG outputs on the address bus.
- The Modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the I-register at the end of each memory access.

  Any M-register can be used with any I-register. The modify value can also be an immediate value instead of an M-register. The size of the modify value must be less than or equal to the length (L-register) of the circular buffer.

- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the I-register.

  L is positive and cannot have a value greater than $2^{32} - 1$. If an L-register's value is zero, its circular buffer operation is disabled.

- The Base (B) register or the B-register plus the L-register is the value with which the DAG compares the modified I-register value after each access.

To address a circular buffer, the DAG steps the Index pointer (I-register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the M-register.

If the Index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (L-register) from the value or adds the length of the buffer to the value, wrapping the Index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer's base address (B-register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Exceptions can be made for video operations. For more information see "Memory Address Alignment" on page 5-13. Circular buffering uses post-modify addressing.

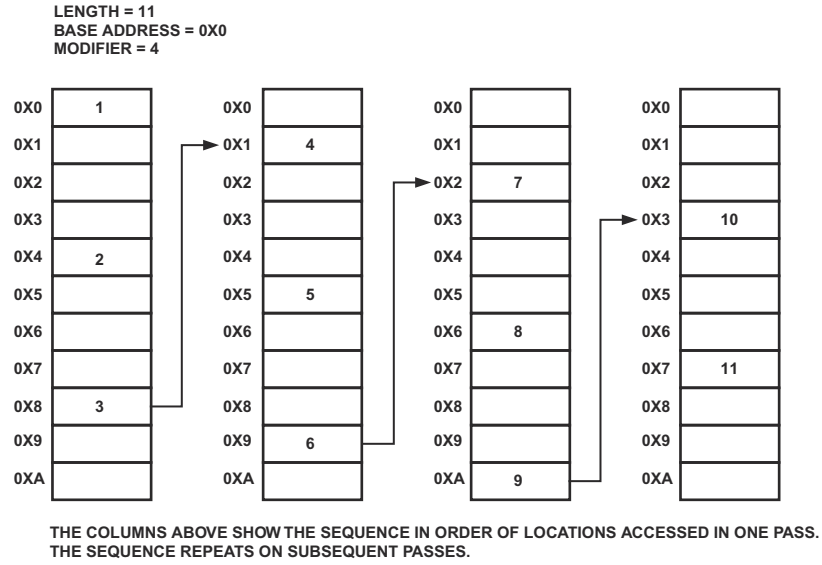


Figure 5-2. Circular Data Buffers

As seen in Figure 5-2, on the first post-modify access to the buffer, the DAG outputs the I-register value on the address bus, then modifies the address by adding the modify value.

- If the updated index value is within the buffer length, the DAG writes the value to the I-register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the L-register value before writing the updated index value to the I-register.

In equation form, these post-modify and wraparound operations work as follows, shown for "I+M" operations.

- If M is positive:

  $I_{new} = I_{old} + M$
  if $I_{old} + M <$ buffer base + length (end of buffer)

  $I_{new} = I_{old} + M - L$
  if $I_{old} + M \geq$ buffer base + length (end of buffer)

- If M is negative:

  $I_{new} = I_{old} + M$
  if $I_{old} + M \geq$ buffer base (start of buffer)

  $I_{new} = I_{old} + M + L$
  if $I_{old} + M <$ buffer base (start of buffer)

Addressing With Bit-reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG's bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the Modify-Increment instruction in *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

Indexed Addressing With Index and Pointer Registers

Indexed addressing uses the value in the Index or Pointer register as an effective address. This instruction can load or store 16- or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the *W* designator is used to preface the load or store.

For example:

```
R0 = [ I2 ];
```

loads a 32-bit value from an address pointed to by `I2` and stores it in the destination register `R0`.

```
R0.H = W [ I2 ];
```

loads a 16-bit value from an address pointed to by `I2` and stores it in the 16-bit destination register `R0.H`.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

```
B [ P1++] = R0 ;
```

stores the 8-bit value from the `R0` register in the address pointed to by the `P1` register, then increments the `P1` register.

Auto-increment and Auto-decrement Addressing

Auto-increment addressing updates the Pointer and Index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the Pointer by 4. A 16-bit word

access updates the Pointer by 2, and an access of an 8-bit word updates the Pointer by 1. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while Index registers may be used only for 16- and 32-bit accesses.

For example:

```
R0 = W [ P1++ ](Z);
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the `P1` Pointer register. The Pointer is then incremented by 2 and the word is zero extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.

For example:

```
R0 = [ I2-- ] ;
```

loads a 32-bit value into the destination register and decrements the Index register by 4.

Pre-modify Stack Pointer Addressing

The only pre-modify instruction in the processor uses the Stack Pointer register, `SP`. The address in `SP` is decremented by 4 and then used as an effective address for the store. The instruction `[ --SP ] = R0` ; is used for stack push operations and can support only a 32-bit word transfer.

Indexed Addressing With Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The Pointer register is modified by the immediate field and then used as the effective address. The value of the Pointer register is not updated.

Alignment exceptions are triggered when a final address is unaligned.

For example, if `P1 = 0x13`, then `[P1 + 0x11]` would effectively be equal to `[0x24]`, which is aligned for all accesses.

Post-modify Addressing

Post-modify addressing uses the value in the Index or Pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other Pointer registers. Index registers are modified by Modify registers. Post-modify addressing does not support the Pointer registers as destination registers, nor does it support byte-addressing.

For example:

```
R5 = [ P1++P2 ] ;
```

loads a 32-bit value into the `R5` register, found in the memory location pointed to by the `P1` register.

The value in the `P2` register is then added to the value in the `P1` register.

For example:

```
R2 = W [ P4++P5 ] (Z) ;
```

loads a 16-bit word into the low half of the destination register `R2` and zero-extends it to 32 bits. The value of the pointer `P4` is incremented by the value of the pointer `P5`.

For example:

```
R2 = [ I2++M1 ];
```

loads a 32-bit word into the destination register `R2`. The value in the Index register, `I2`, is updated by the value in the Modify register, `M1`.

Modifying DAG and Pointer Registers

The DAGs support operations that modify an address value in an Index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The address-modify operation modifies addresses in any DAG Index and Pointer register (`I[3:0]`, `P[5:0]`, `FP`, `SP`) without accessing memory. If the Index register's corresponding B- and L-registers are set up for circular buffering, the address-modify operation performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (index += modifier). For Index registers, an M-register is used as the modifier. For Pointer registers, another P-register is used as the modifier.

Consider the example, `I1 += M2` ;

This instruction adds `M2` to `I1` and updates `I1` with the new value.

Memory Address Alignment

The processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions—for example, many of the Video ALU instructions—automatically disable alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the `DISALGNEXPT` instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are b#00.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be b#0.

Table 5-1 summarizes the types of transfers and transfer sizes supported by the addressing modes.

Table 5-1. Types of Transfers Supported and Transfer Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed	To and from Data Registers	LOADS: 32-bit word 16-bit, zero extended half word 16-bit, sign extended half word 8-bit, zero extended byte 8-bit, sign extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data register high half 16-bit half word to Data register low half 16-bit, zero extended half word 16-bit, sign extended half word STORES: 32-bit word 16-bit half word from Data register high half 16-bit half word from Data register low half

Be careful when using the DISALGNEXPT instruction, because it disables automatic detection of memory alignment errors. The DISALGNEXPT instruction only affects misaligned loads that use I-register indirect addressing. Misaligned loads using P-register addressing will still cause an exception.

Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 5-2. Addressing Modes

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc [I0++M0]	*				*		

DAG Instruction Summary

Table 5-3 lists the DAG instructions. For more information on assembly language syntax, see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. In Table 5-3, note the meaning of these symbols:

- Dreg denotes any Data register File register.
- Dreg_lo denotes the lower 16 bits of any Data register File register.
- Dreg_hi denotes the upper 16 bits of any Data register File register.
- Preg denotes any Pointer register, `FP`, or `SP` register.
- Ireg denotes any DAG Index register.
- Mreg denotes any DAG Modify register.
- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bits wide, immediate value.
- uimmAmB denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.

The *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* more fully describes the options that may be applied to these instructions and the sizes of immediate fields.

DAG instructions do not affect the `ASTAT` register flags.

Table 5-3. DAG Instruction Summary

Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg – uimm17m4] ;
Preg = [FP – uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg – uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP – uimm7m4] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg – uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg – uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg_hi = W [Ireg] ;
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;
Dreg = B [Preg – uimm15] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Dreg = B [Preg – uimm15] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg – uimm17m4] = Preg ;
[FP – uimm7m4] = Preg ;
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg – uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;
[FP – uimm7m4] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [Ireg] = Dreg_lo ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;
W [Preg + uimm16m2] = Dreg ;
W [Preg – uimm16m2] = Dreg ;
W [Preg ++ Preg] = Dreg_lo ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg – uimm15] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg = Preg << 2 ;
Preg = Preg >> 2 ;
Preg = Preg >> 1 ;
Preg = Preg + Preg << 1 ;
Preg = Preg + Preg << 2 ;
Preg –= Preg ;
Ireg –= Mreg ;

6 MEMORY

The processor supports a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories are located on the chip and are faster than the Level 2 (L2) memory systems. The Level 2 (L2) memories are off-chip and have longer access latencies. The faster L1 memories, which are typically small scratchpad memory or cache memories, are found within the core itself.

Memory Architecture

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

- L1 static random access memories (SRAM)
- a set of memory-mapped registers (MMRs)
- a boot read-only memory (ROM)

A portion of the internal L1 SRAM can also be configured to run as cache. The processor also provides support for an external memory space that includes asynchronous memory space and synchronous DRAM (SDRAM) space. See Chapter 18, “External Bus Interface Unit” for a detailed discussion of each of these memory regions and the controllers that support them.

Figure 6-1 on page 6-3 provides an overview of the ADSP-BF539 processor system memory map. Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

The memory configuration for the ADSP-BF539 processors is shown in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in Supervisor or Emulation mode (see Chapter 3, "Operating Modes and States".

The lowest 1K byte of internal memory space is occupied by the boot ROM. Depending on the booting option selected, the appropriate boot program is executed from this memory space when the processor is reset (see "Booting Methods" on page 3-18.)

Within the external memory map, four banks of asynchronous memory space and one bank of SDRAM memory are available. Each of the asynchronous banks is 1M byte and the SDRAM bank is up to 128M byte.

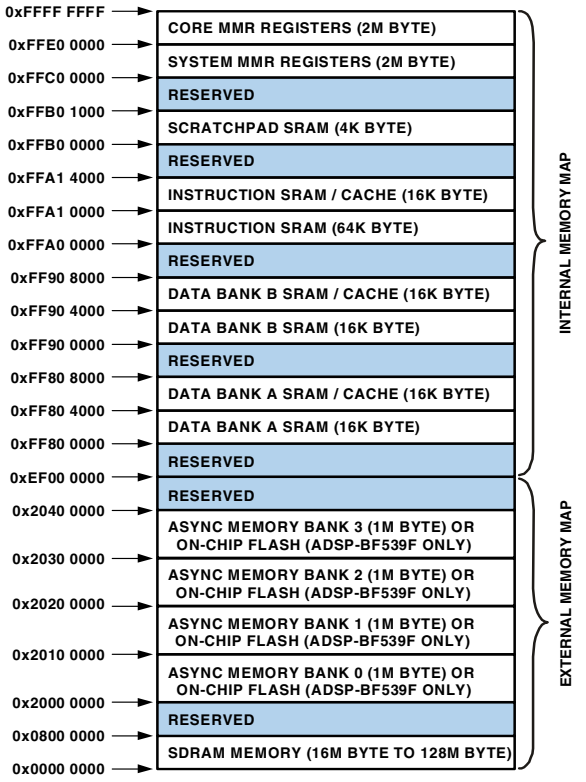


Figure 6-1. ADSP-BF539/ADSP-BF539F Memory Map

Overview of Internal Memory

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

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

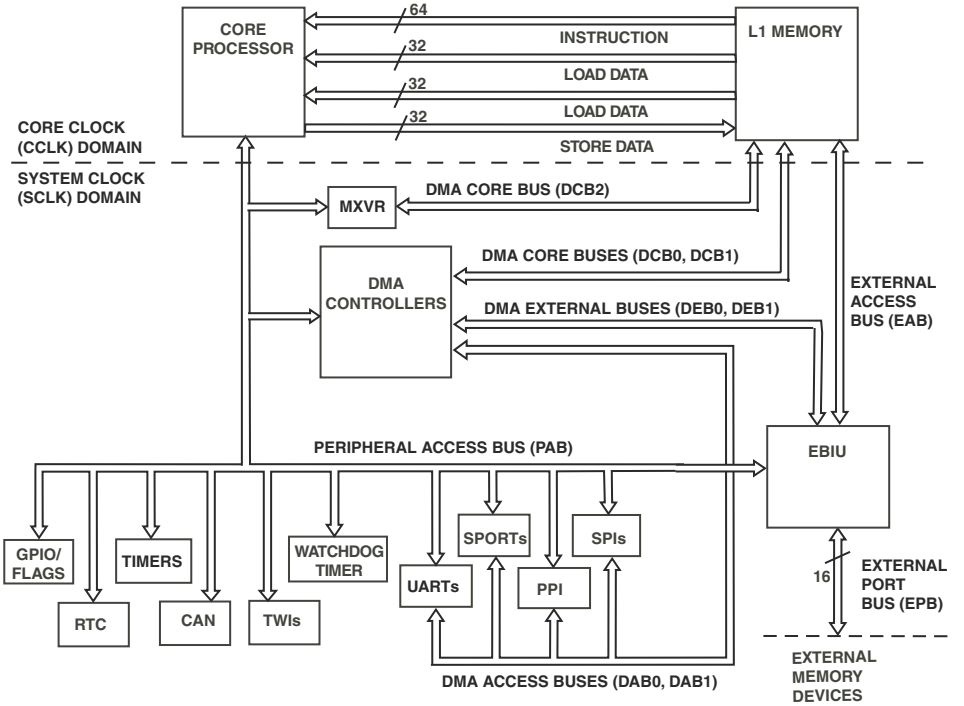


Figure 6-2. Processor Memory Architecture

The L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM access at processor clock rate (CCLK) for critical DSP algorithms and fast context switching
- Instruction and data cache options for microcontroller code, excellent High Level Language (HLL) support, and ease of programming cache control instructions, such as PREFETCH and FLUSH
- Memory protection

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

Overview of Scratchpad Data SRAM

The processor provides a dedicated 4K byte bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache or targeted by DMA. Typical applications use the scratchpad data memory where speed is critical. For example, the User and Supervisor stacks should be mapped to the scratchpad memory for the fastest context switching during interrupt handling.

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

Scratchpad data SRAM cannot be accessed by the DMA controller.

L1 Instruction Memory

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

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

L1 instruction memory can be used only to store instructions.

IMEM_CONTROL Register

The instruction memory control register (IMEM_CONTROL) contains control bits for the L1 instruction memory. By default after reset, cache and Cacheability Protection Lookaside Buffer (CPLB) address checking is disabled (see "L1 Instruction Cache" on page 6-11).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRUPRIO bits (see "Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51) are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPLB_LRUPRIO bits, and then on LRU (least recently used). See "Instruction Cache Locking by Line" on page 6-17 for complete details. This bit must be 0 to allow the state of the CPLB_LRUPRIO bits to be stored when new lines are cached.

The ILOC[3:0] bits provide a useful feature only after code has been manually loaded into cache. See "Instruction Cache Locking by Way" on page 6-18. These bits specify which Ways to remove from the cache replacement policy. This has the effect of locking code present in nonpar-

ticipating Ways. Code in nonparticipating Ways can still be removed from the cache using an IFLUSH instruction. If an ILOC[3:0] bit is 0, the corresponding Way is not locked and that Way participates in cache replacement policy. If an ILOC[3:0] bit is 1, the corresponding Way is locked and does not participate in cache replacement policy.

The IMC bit reserves a portion of L1 instruction SRAM to serve as cache. Note reserving memory to serve as cache will not alone enable L2 memory accesses to be cached. CPLBs must also be enabled using the EN_ICPLB bit and the CPLB descriptors (ICPLB_DATAx and ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.

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

- Reserved (nonpopulated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using this bit prior to updating their descriptors (DCPLB_DATAx and DCPLB_ADDRx registers). Note since load store ordering is weak (see "Ordering of Loads and Stores" on page 6-65), disabling of CPLBs should be proceeded by a CSYNC.

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

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

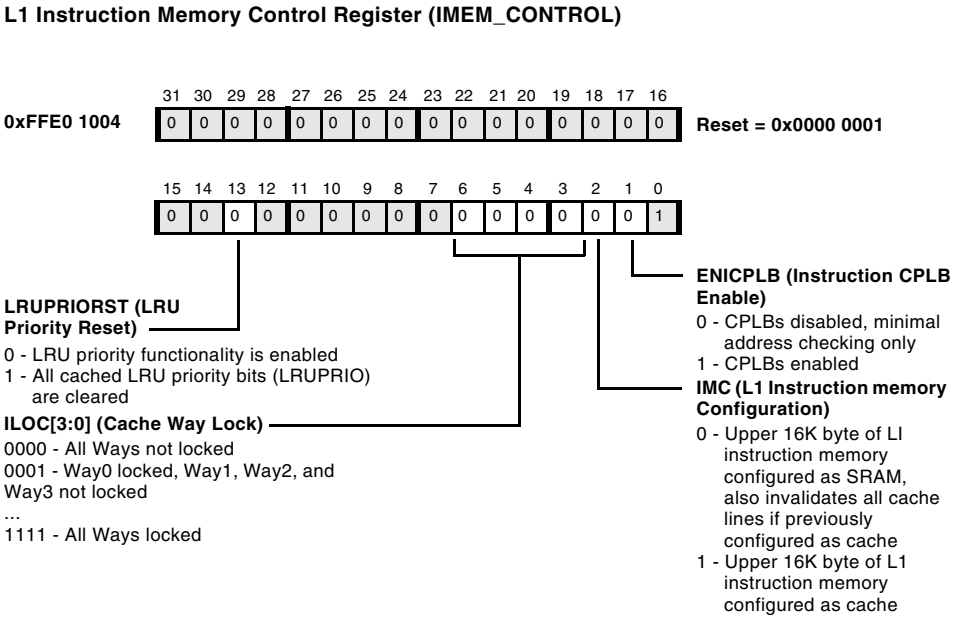


Figure 6-3. L1 Instruction Memory Control Register

L1 Instruction SRAM

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

The DAGs, which are described in Chapter 5, cannot access L1 instruction memory directly. A DAG reference to instruction memory SRAM space generates an exception (see “Exceptions” on page 4-45).

Write access to the L1 instruction SRAM memory must be made through the 64-bit wide system DMA port. Because the SRAM is implemented as a collection of single ported subbanks, the instruction memory is effectively dual ported.

Table 6-1 lists the memory start locations of the L1 instruction memory subbanks.

Table 6-1. L1 Instruction Memory Subbanks

Memory Subbank	Memory Start Location
0	0xFFA0 0000
1	0xFFA0 1000
2	0xFFA0 2000
3	0xFFA0 3000
4	0xFFA0 4000
5	0xFFA0 5000
6	0xFFA0 6000
7	0xFFA0 7000
8	0xFFA0 8000
9	0xFFA0 9000
10	0xFFA0 A000
11	0xFFA0 B000
12	0xFFA0 C000
13	0xFFA0 D000
14	0xFFA0 E000
15	0xFFA0 F000

Figure 6-4 on page -10 describes the bank architecture of the L1 instruction memory. As the figure shows, each 16K byte bank is made up of four 4K byte subbanks.

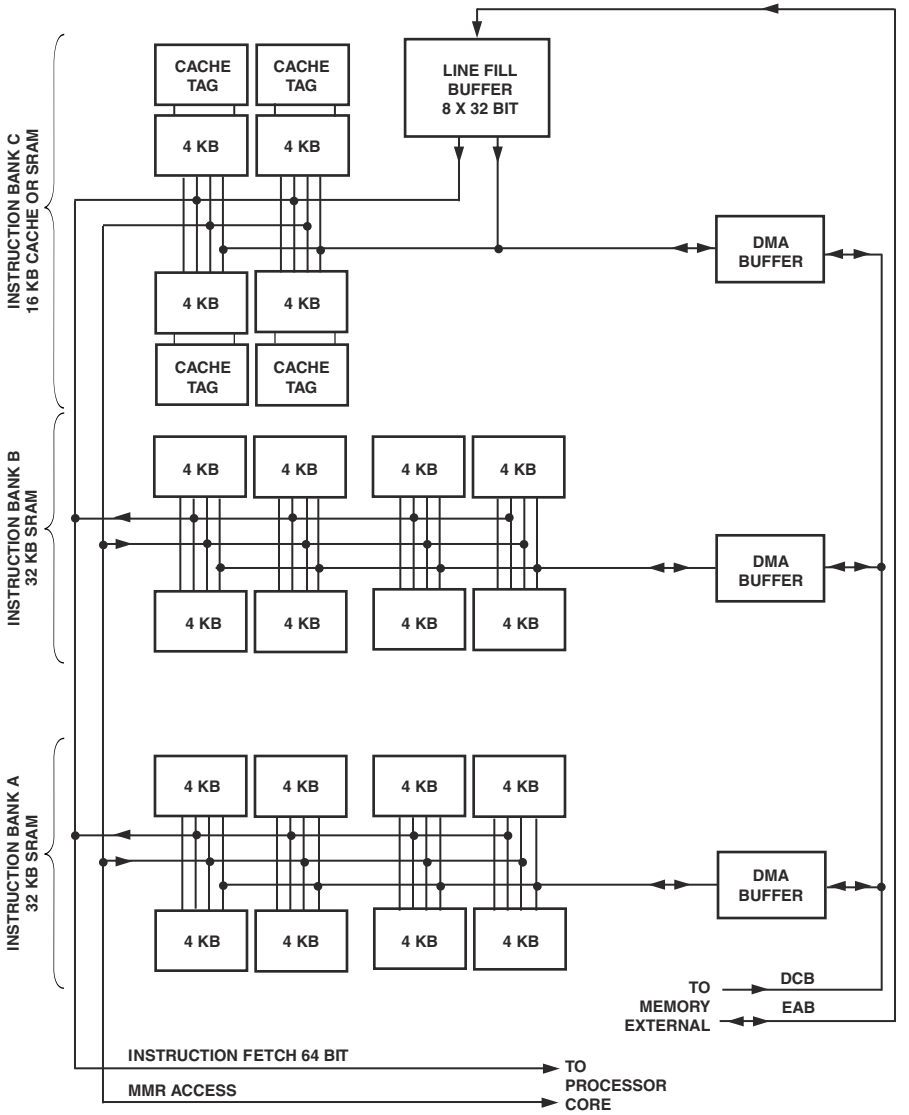


Figure 6-4. L1 Instruction Memory Bank Architecture

L1 Instruction Cache

For information about cache terminology, see "Terminology" on page 6-72.

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

When cache is enabled, only memory pages further specified as cacheable by the CPLBs will be cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in "Memory Protection and Properties" on page 6-43.

Figure 6-5 on page -13 shows the overall Blackfin processor instruction cache organization.

Cache Lines

As shown in Figure 6-5, the cache consists of a collection of cache lines. Each cache line is made up of a *tag* component and a *data* component.

- The tag component incorporates a 20-bit address tag, least recently used (LRU) bits, a Valid bit, and a Line Lock bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

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

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

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

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

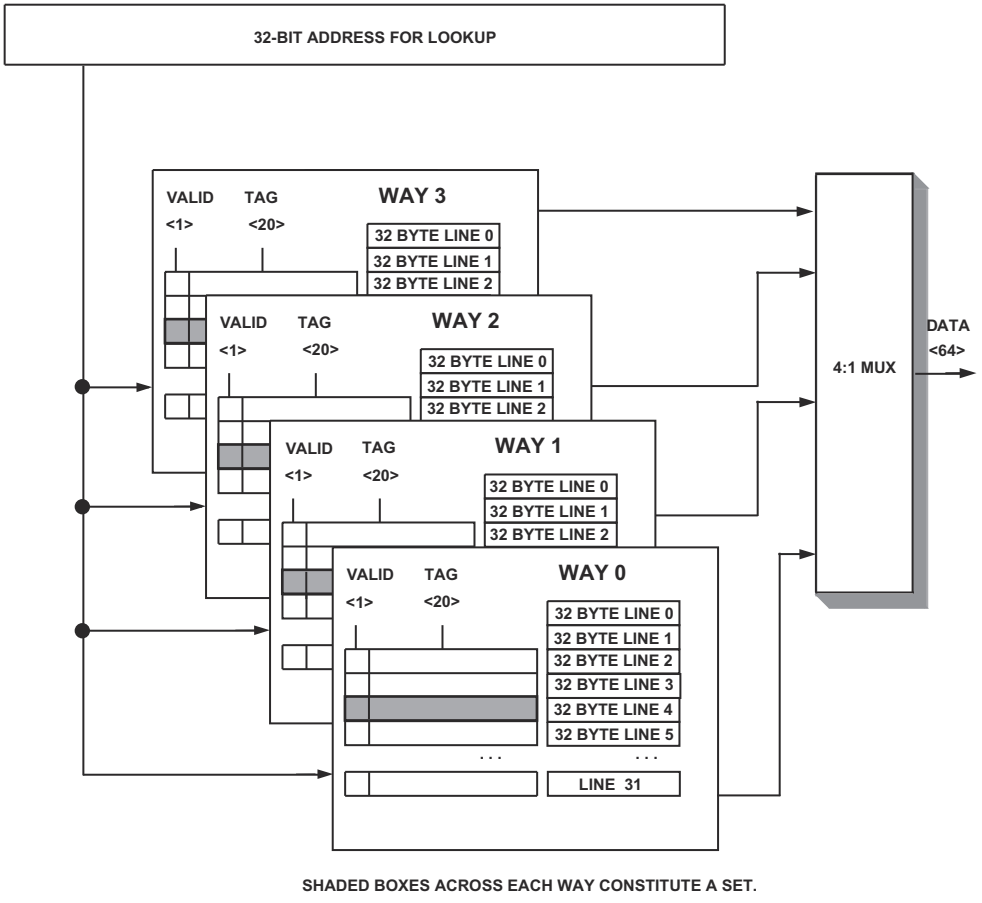


Figure 6-5. Blackfin Processor Instruction Cache Organization

The tag and data components of a cache line are illustrated in Figure 6-6.

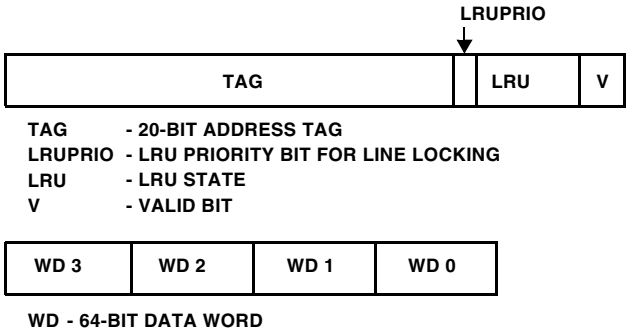


Figure 6-6. Cache Line – Tag and Data Portions

Cache Hits and Misses

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

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

Cache Line Fills

A cache line fill consists of fetching 32 bytes of data from memory. The operation starts when the instruction memory unit requests a line-read data transfer (a burst of four 64-bit words of data) on its external read-data port. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction word first. After it has returned the target instruction word, the next three words are fetched in sequential address order. This fetch wraps around if necessary, as shown in Table 6-2.

Table 6-2. Cache Line Word Fetching Order

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

Line Fill Buffer

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

Cache Line Replacement

When the instruction memory unit is configured as cache, bits 9 through 5 of the instruction fetch address are used as the index to select the cache set for the tag-address compare operation. If the tag-address compare operation results in a cache miss, the Valid and LRU bits for the selected

set are examined by a cache line replacement unit to determine the entry to use for the new cache line, that is, whether to use way0, way1, way2, or way3. See Figure 6-5 on page 6-13.

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

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

For example:

- If way3 is invalid and ways 0, 1, 2 are valid, way3 is selected for the new cache line.
- If ways 0 and 1 are invalid and ways 2 and 3 are valid, way0 is selected for the new cache line.
- If ways 2 and 3 are invalid and ways0 and 1 are valid, way2 is selected for the new cache line.

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

Instruction Cache Management

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

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

See "Instruction Cache Invalidation" on page 6-19.

Instruction Cache Locking by Line

The CPLB_LRUPRIO bits in the ICPLB_DATAx registers (see "Memory Protection and Properties" on page 6-43) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line's tag. It is then used in conjunction with the LRU (least recently used) policy to determine which Way is victimized when all cache Ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either "low" or "high" importance. In a modified LRU policy, a high can replace a low, but a low cannot replace a high. If all Ways are occupied by highs, an otherwise cacheable low will still be fetched for the core, but will not be cached. Fetched highs seek to replace unoccupied Ways first, then least recently used lows next, and finally other highs using the LRU policy. Lows can only replace unoccupied Ways or other lows, and do so using the LRU policy. If *all* previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the LRUPRIRST bit in the IMEM_CONTROL register (see page -6).

Instruction Cache Locking by Way

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

An example sequence is provided below to demonstrate how to lock down way0:

- If the code of interest may already reside in the instruction cache, invalidate the entire cache first (for an example, see "Instruction Cache Management" on page 6-17).
- Disable interrupts, if required, to prevent interrupt service routines (ISRs) from potentially corrupting the locked cache.
- Set the locks for the other ways of the cache by setting ILOC[3:1]. Only way0 of the instruction cache can now be replaced by new code.
- Execute the code of interest. Any cacheable exceptions, such as exit code, traversed by this code execution are also locked into the instruction cache.
- Upon exit of the critical code, clear ILOC[3:1] and set ILOC[0]. The critical code (and the instructions which set ILOC[0]) is now locked into way0.
- Re-enable interrupts, if required.

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

Instruction Cache Invalidation

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

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

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

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

For invalidating the complete instruction cache, a third method is available. By clearing the IMC bit in the IMEM_CONTROL register (see Figure 6-3 on page 6-8), all Valid bits in the instruction cache are set to the invalid state. A second write to the IMEM_CONTROL register to set the IMC bit configures the instruction memory as cache again. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Instruction Test Registers

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

When the instruction test command register (ITEST_COMMAND) is used, the L1 cache data or tag arrays are accessed, and data is transferred through the instruction test data registers (ITEST_DATA[1:0]). The ITEST_DATAx registers contain either the 64-bit data that the access is to write to or the 64-bit data that was read during the access. The lower 32 bits are stored in the ITEST_DATA[0] register, and the upper 32 bits are stored in the ITEST_DATA[1] register. When the tag arrays are accessed, ITEST_DATA[0] is used. Graphical representations of the ITEST registers begin with Figure 6-7 on page -21.

The following figures describe the ITEST registers:

- Figure 6-7 on page 6-21
- Figure 6-8 on page 6-22
- Figure 6-9 on page 6-23

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

Instruction Test Command (ITEST_COMMAND) Register

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

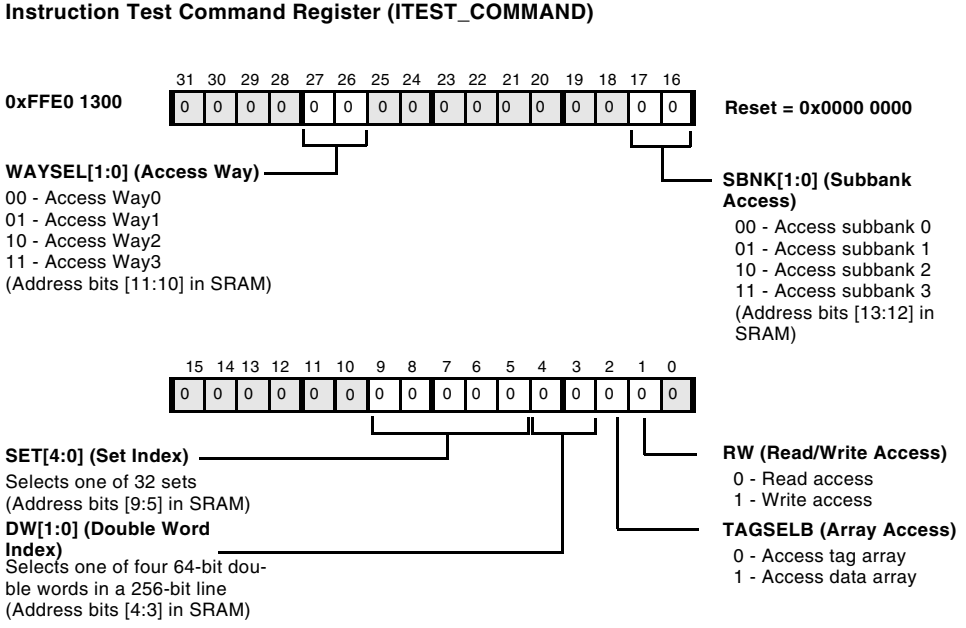


Figure 6-7. Instruction Test Command Register

Instruction Test Data (ITEST_DATA1) Register

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

Instruction Test Data 1 Register (ITEST_DATA1)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 6-11.

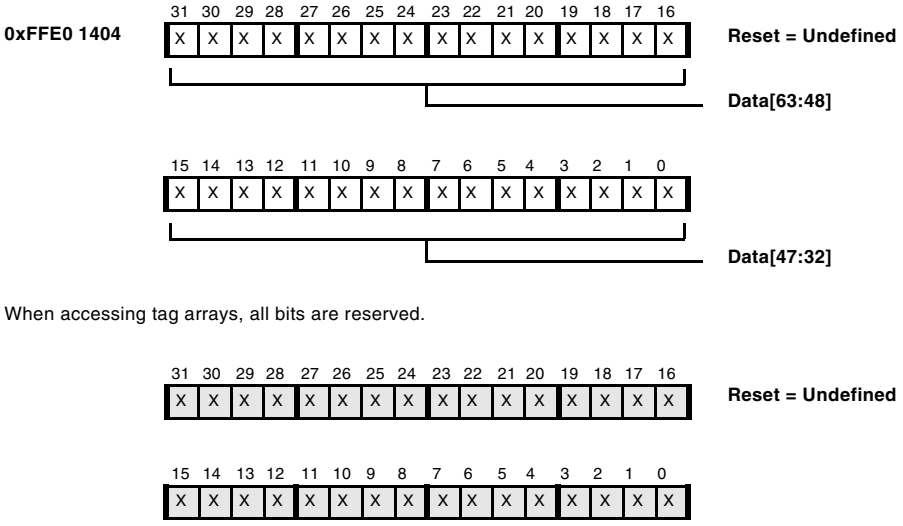


Figure 6-8. Instruction Test Data 1 Register

Instruction Test Data 0 (ITEST_DATA0) Register

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

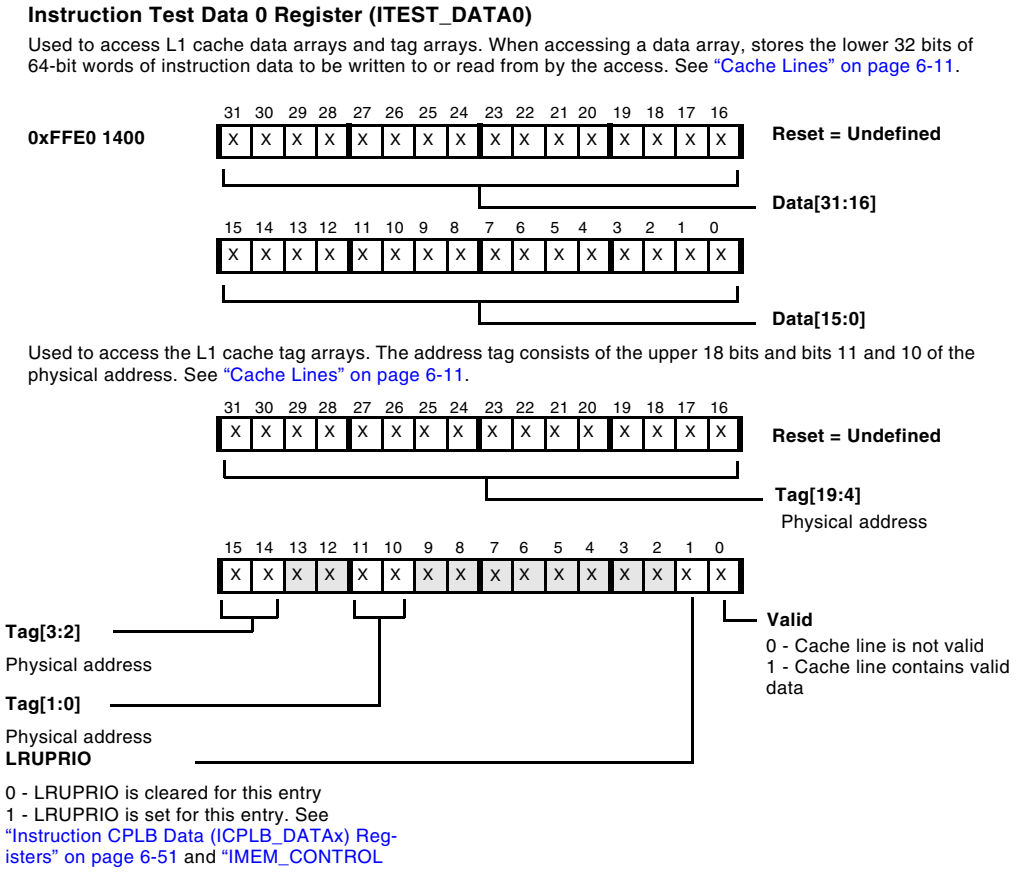


Figure 6-9. Instruction Test Data 0 Register

L1 Data Memory

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

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

L1 data memory can be used only to store data.

Data Memory Control (DMEM_CONTROL) Register

The data memory control register (DMEM_CONTROL) contains control bits for the L1 data memory.

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

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

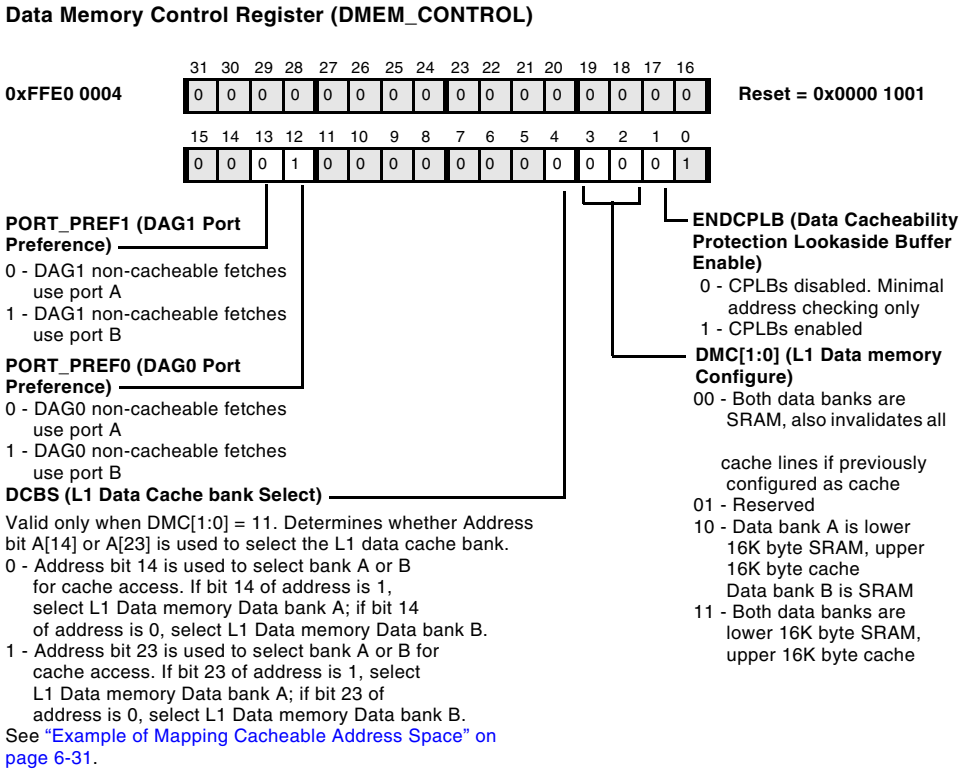


Figure 6-10. L1 Data Memory Control Register

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

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

The ENDCPLB bit is used to enable/disable the 16 Cacheability Protection Lookaside Buffers (CPLBs) used for data (see "L1 Data Cache" on page 6-30). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in User mode

CPLBs must be disabled using this bit prior to updating their descriptors (registers DCPLB_DATAx and DCPLB_ADDRx). Note that since load store ordering is weak (see "Ordering of Loads and Stores" on page 6-65), disabling CPLBs should be preceded by a CSYNC instruction.

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

By default after reset, all L1 data memory serves as SRAM. The DMC[1:0] bits can be used to reserve portions of this memory to serve as cache instead. Reserving memory to serve as cache does not enable L2 memory accesses to be cached. To do this, CPLBs must also be enabled (using the ENDCPLB bit) and CPLB descriptors (registers DCPLB_DATAx and DCPLB_ADDRx) must specify chosen memory pages as cache-enabled.

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

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

L1 Data SRAM

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

Table 6-3 shows how the subbank organization is mapped into memory.

Table 6-3. L1 Data Memory SRAM Subbank Start Addresses

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

Table 6-3. L1 Data Memory SRAM Subbank Start Addresses (Cont'd)

Memory Bank and Subbank	Start Address
Data bank B, Subbank 3	0xFF90 3000
Data bank B, Subbank 4	0xFF90 4000
Data bank B, Subbank 5	0xFF90 5000
Data bank B, Subbank 6	0xFF90 6000
Data bank B, Subbank 7	0xFF90 7000

Figure 6-11 shows the L1 data memory architecture.

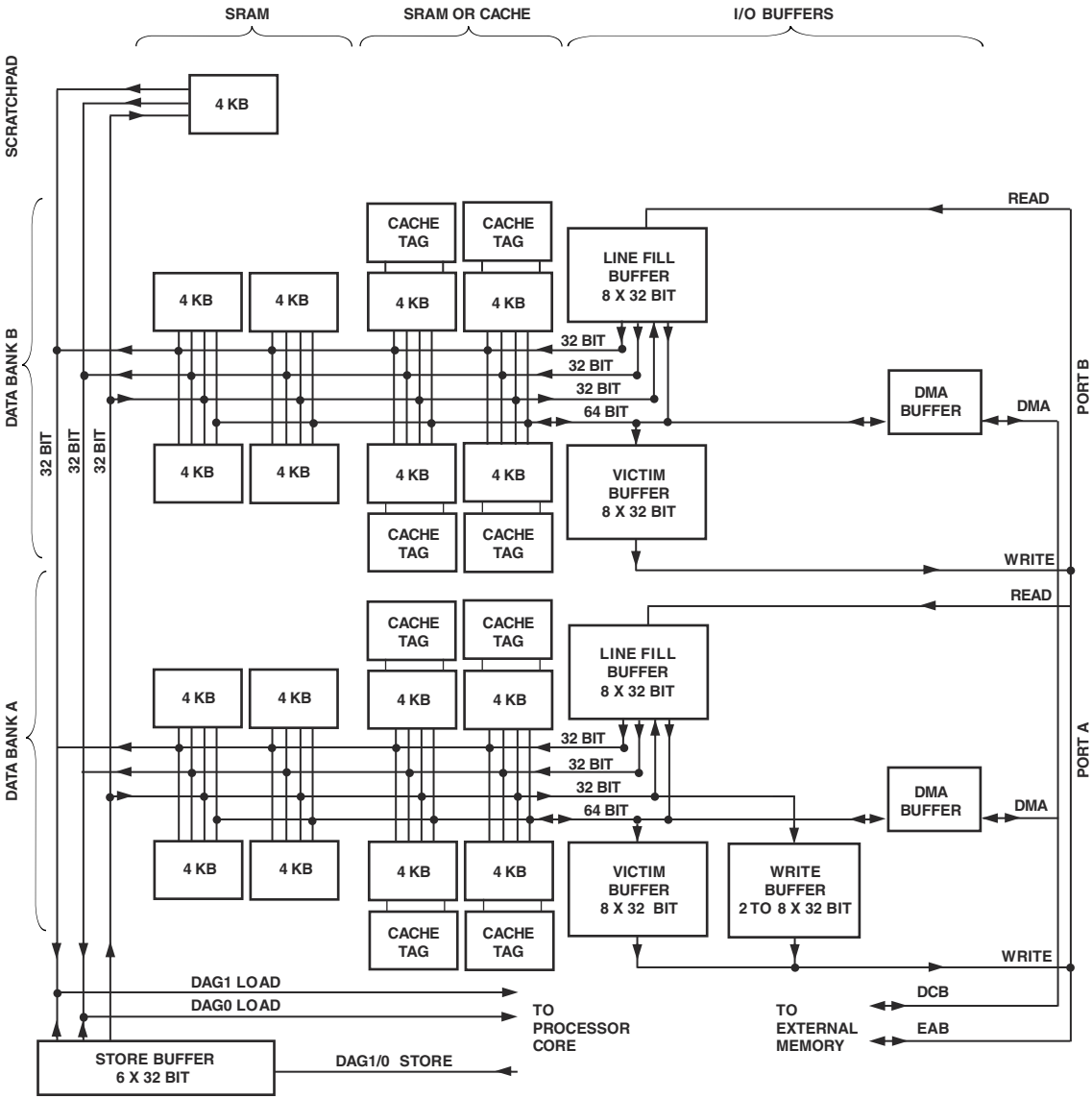


Figure 6-11. L1 Data Memory Architecture

L1 Data Cache

For definitions of cache terminology, see "Terminology" on page 6-72.

When data cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), either 16K byte of data bank A or 16K byte of both data bank A and data bank B can be set to serve as cache. When configured as cache memory for the processor, the upper 16K bytes of the bank are used. Unlike instruction cache, which is 4-way set associative, data cache is 2-way set associative. When two banks are available and enabled as cache, additional sets rather than ways are created. When both data bank A and data bank B have memory serving as cache, the DCBS bit in the DMEM_CONTROL register may be used to control which half of all address space is handled by which bank of cache memory. The DCBS bit selects either address bit 14 or 23 to steer traffic between the cache banks. This provides some control over which addresses alias into the same set. It may therefore be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Accesses to cache do not collide unless they are to the same 4K byte subbank, the same half bank, and to the same bank. Cache has less apparent multi-ported behavior than SRAM due to the overhead in maintaining tags. When cache addresses collide, access is granted first to the DTEST register accesses, then to the store buffer, and finally to cache fill/victim traffic.

Three different cache modes are available.

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

Cache mode is selected by the DCPLB descriptors (see "Memory Protection and Properties" on page 6-43). Any combination of these cache modes can be used simultaneously since cache mode is selectable for each memory page independently.

If cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), data CPLBs should also be enabled (controlled by ENDCPLB bit in the DMEM_CONTROL register). Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

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

Example of Mapping Cacheable Address Space

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

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

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

- If DCBS = 0, then A[14] is part of the address index, and all addresses in which A[14] = 0 use Data bank B. All addresses in which A[14] = 1 use Data bank A.

  In this case, A[23] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

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

  In this case, A[14] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

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

- If DCBS = 0, A[14] selects Data bank A instead of Data bank B.

  Alternating 16K byte pages of memory map into each of the two 16K byte caches implemented by the two data banks.

  As a result, the cache operates as a single, contiguous, 2-way set associative 32K byte cache. Each way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each way).

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

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

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

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

  With DCBS = 1, the system functions more like two independent caches, each a 2-Way set associative 16K byte cache. Each bank serves an alternating set of 8M byte blocks of memory.

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

For example, if the application is working from a data set that is 1M byte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by data bank B (a 2-Way set associative 16K byte cache). In this instance, the application never derives any benefit from data bank A.

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

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

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

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

Figure 6-12 shows an example of how mapping is performed when `DCBS = 1`.

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

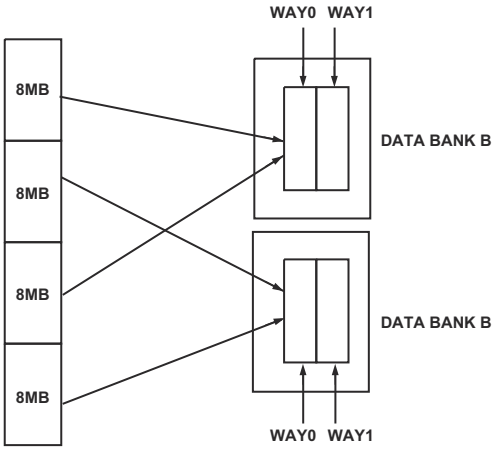


Figure 6-12. Data Cache Mapping When DCBS = 1

Data Cache Access

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

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

- If the state of the line is exclusive (clean), the new tag and data write over the old line.
- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data.

  If the line is dirty, the current contents of the cache are copied back to external memory before the new data is written to the cache.

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

The data cache supports hit-under-a-store miss, and hit-under-a-prefetch miss. In other words, on a write-miss or execution of a `PREFETCH` instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a 4-cycle stall. Furthermore, a subsequent load or store instruction can hit in the L1 cache while the line fill completes.

Interrupts of sufficient priority (relative to the current context) cancel a stalled load instruction. Consequently, if the load operation misses the L1 data memory cache and generates a high latency line fill operation on the system interface, it is possible to interrupt the core, causing it to begin processing a different context. The system access to fill the cache line is not cancelled, and the data cache is updated with the new data before any further cache miss operations to the respective data bank are serviced. For more information see "Exceptions" on page 4-45.

Cache Write Method

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

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache.

  If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.

- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line.

The L1 data memory employs a full cache line width copyback buffer on each data bank. In addition, a two-entry write buffer in the L1 data memory accepts all stores with cache inhibited or store-through protection. An `SSYNC` instruction flushes the write buffer.

Interrupt Priority Register and Write Buffer Depth

The interrupt priority register (`IPRIO`) can be used to control the size of the write buffer on port A (see "L1 Data Memory Architecture" on page 6-29).

The `IPRIO[3:0]` bits can be programmed to reflect the low priority interrupt watermark. When an interrupt occurs, causing the processor to vector from a low priority interrupt service routine to a high priority interrupt service routine, the size of the write buffer increases from two to eight 32-bit words deep. This allows the interrupt service routine to run and post writes without an initial stall, in the case where the write buffer was already filled in the low priority interrupt routine. This is most useful when posted writes are to a slow external memory device. When returning from a high priority interrupt service routine to a low priority interrupt service routine or user mode, the core stalls until the write buffer has completed the necessary writes to return to a two-deep state. By default, the write buffer is a fixed two-deep FIFO.

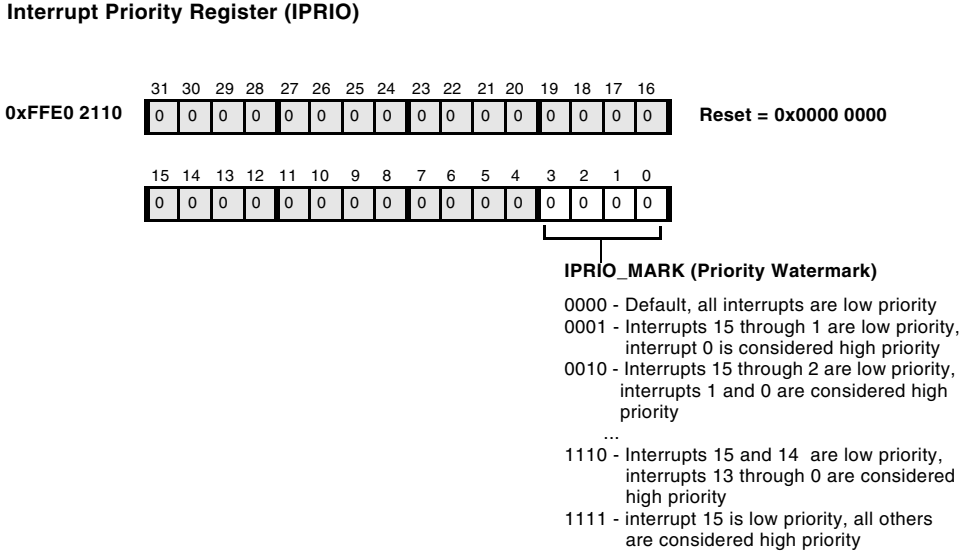


Figure 6-13. Interrupt Priority Register

Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in User and Supervisor modes. The instructions are PREFETCH, FLUSH, and FLUSHINV.

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

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

Data Cache Invalidation

Besides the FLUSHINV instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required. The first technique directly invalidates valid bits by setting the invalid bit of each cache line to the invalid state. To implement

this technique, additional MMRs (DTEST_COMMAND and DTEST_DATA[1:0]) are available to allow arbitrary reads/writes of all the cache entries directly. This method is explained in the next section.

For invalidating the complete data cache, a second method is available. By clearing the DMC[1:0] bits in the DMEM_CONTROL register (see Figure 6-10 on page 6-25), all valid bits in the data cache are set to the invalid state. A second write to the DMEM_CONTROL register to set the DMC[1:0] bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Data Test Registers

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

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

A CSYNC instruction is required after writing the DTEST_COMMAND MMR.

Figure 6-14 through Figure 6-16 describe the DTEST registers.

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

Data Test Command (DTEST_COMMAND) Register

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

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

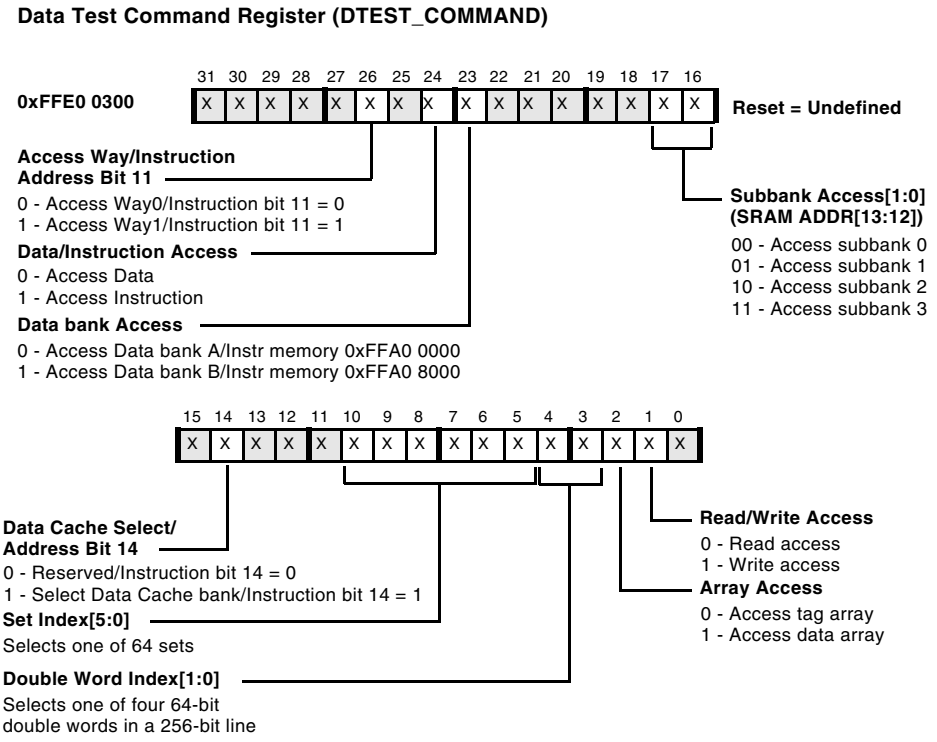


Figure 6-14. Data Test Command Register

Data Test Data (DTEST_DATA1) Register

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

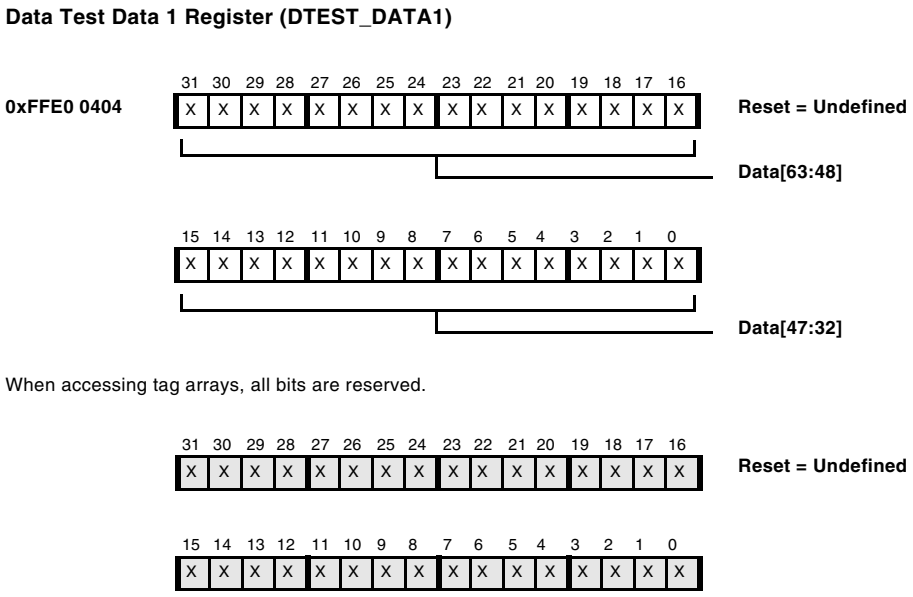


Figure 6-15. Data Test Data 1 Register

Data Test Data (DTEST_DATA0) Register

The data test data 0 register (DTEST_DATA0) stores the lower 32 bits of the 64-bit data to be written, or it contains the lower 32 bits of the destination for the 64-bit data read. The DTEST_DATA0 register is also used to access the tag arrays and contains the valid and dirty bits, which indicate the state of the cache line.

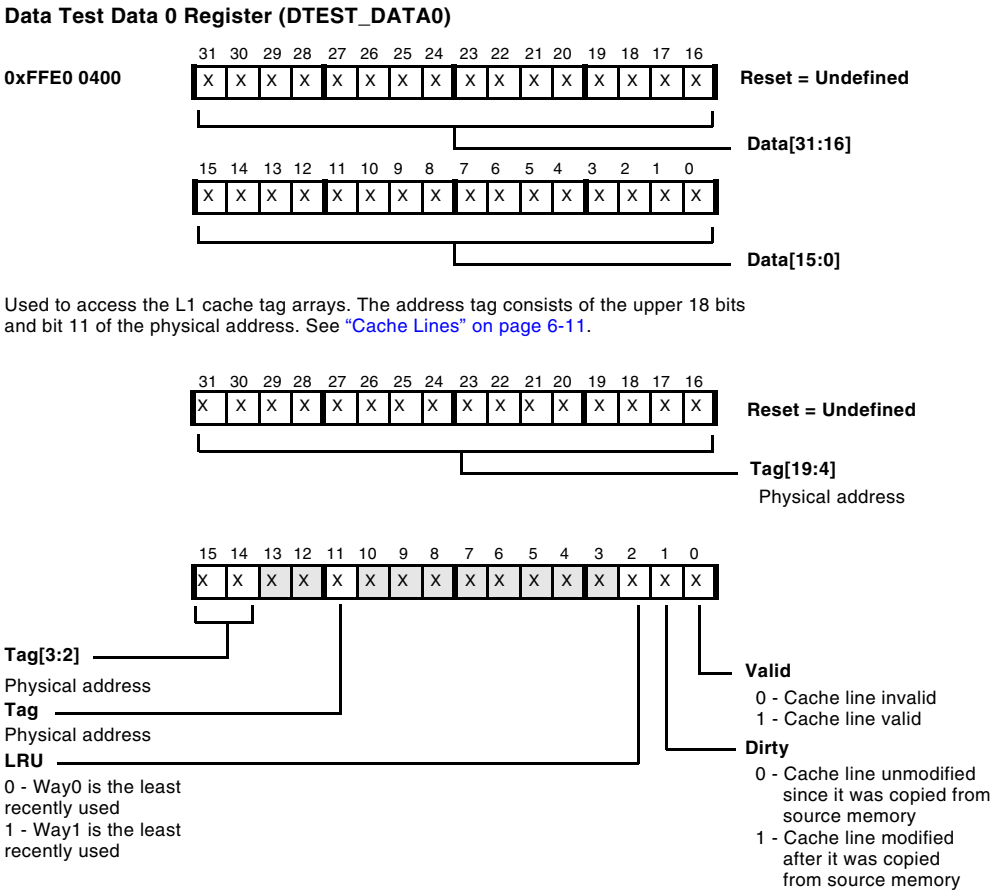


Figure 6-16. Data Test Data 0 Register

External Memory

The external memory space is shown in Figure 6-1 on page -3. One of the memory regions is dedicated to SDRAM support. The size of the SDRAM bank is programmable and can range in size from 16M byte to 128M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 1M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000. For the ADSP-BF539F processors, the on-chip flash memory can be mapped to any of these four banks of asynchronous memory.

On-Chip Flash Memory

The ADSP-BF539F processors provide on-chip flash memory options. This flash memory is a separate die inside the package, and it can be mapped to any of the four asynchronous memory banks by connecting the $\overline{\texttt{FCE}}$ pin to the appropriate $\overline{\texttt{AMSx}}$ pin. If the $\overline{\texttt{FCE}}$ pin is connected to the $\overline{\texttt{AMS0}}$ pin and the processor is configured to boot from asynchronous memory, the processor will boot from the on-chip flash memory. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for details regarding the on-chip flash memory.

Memory Protection and Properties

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

Memory Management Unit

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

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

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries are used for instruction fetch requests; these are called *ICPLBs*. Another sixteen CPLB entries are used for data transactions; these are called *DCPLBs*. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 instruction memory control (`IMEM_CONTROL`) and L1 data memory control (`DMEM_CONTROL`) registers, respectively. These registers are shown in Figure 6-3 on page 6-8 and Figure 6-10 on page 6-25, respectively.

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

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

For data operations:

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

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

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

Memory Pages

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

- 1K byte
- 4K byte

- 1M byte
- 4M byte

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

Memory Page Attributes

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

- Page size

  1K byte, 4K byte, 1M byte, 4M byte

- Cacheable/non-cacheable

  Accesses to this page use the L1 cache or bypass the cache.

- If cacheable: write-through/write-back

  Data writes propagate directly to memory or are deferred until the cache line is reallocated. If write-through, allocate on read only, or read and write.

- Dirty/modified

  The data in this page in memory has changed since the CPLB was last loaded.

- Supervisor write access permission

– Enables or disables writes to this page when in Supervisor mode.
– Data pages only.

- User write access permission

  – Enables or disables writes to this page when in User mode.
  – Data pages only.

- User read access permission

  Enables or disables reads from this page when in User mode.

- Valid

  Check this bit to determine whether this is valid CPLB data.

- Lock

  Keep this entry in MMR; do not participate in CPLB replacement policy.

Page Descriptor Table

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

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

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than will fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a page descriptor table, is used; in it can be stored all

the potentially required CPLB descriptors. The specific format for the page descriptor table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement page descriptor table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

CPLB Management

When the Blackfin processor issues a memory operation for which no valid CPLB (cacheability protection lookaside buffer) descriptor exists in an MMR pair, an exception occurs that places the processor into Supervisor mode and vectors to the MMU exception handler (see "Exceptions" on page 4-45 for more information). The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

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

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

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

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

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

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

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

MMU Application

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

If all L1 memory is configured as SRAM, then the data and instruction MMU functions are optional, depending on the application's need for protection of memory spaces either between tasks or between User and Supervisor modes. To protect memory between tasks, the operating system can maintain separate tables of instruction and/or data memory pages available for each task and make those pages visible only when the relevant

task is running. When a task switch occurs, the operating system can ensure the invalidation of any CPLB descriptors on chip that should not be available to the new task. It can also preload descriptors appropriate to the new task.

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

If either the L1 instruction memory or the L1 data memory is configured partially or entirely as cache, the corresponding CPLBs must be enabled. When an instruction generates a memory request and the cache is enabled, the processor first checks the ICPLBs to determine whether the address requested is in a cacheable address range. If no valid ICPLB entry in an MMR pair corresponds to the requested address, an MMU exception is generated to obtain a valid ICPLB descriptor to determine whether the memory is cacheable or not. As a result, if the L1 instruction memory is enabled as cache, then any memory region that contains instructions must have a valid ICPLB descriptor defined for it. These descriptors must either reside in MMRs at all times or be resident in a memory-based page descriptor table that is managed by the MMU exception handler. Likewise, if either or both L1 data banks are configured as cache, all potential data memory ranges must be supported by DCPLB descriptors.

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

Examples of Protected Memory Regions

In Figure 6-17, a starting point is provided for basic CPLB allocation for Instruction and Data CPLBs. Note some ICPLBs and DCPLBs have common descriptors for the same address space.

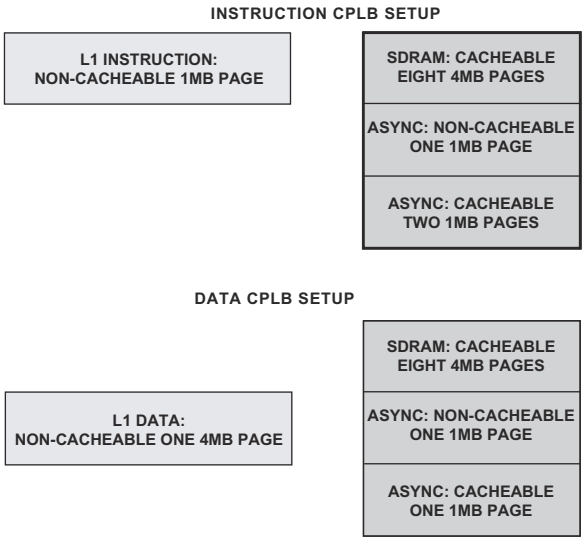


Figure 6-17. Examples of Protected Memory Regions

Instruction CPLB Data (ICPLB_DATAx) Registers

Figure 6-18 describes the ICPLB data registers (`ICPLB_DATAx`).

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

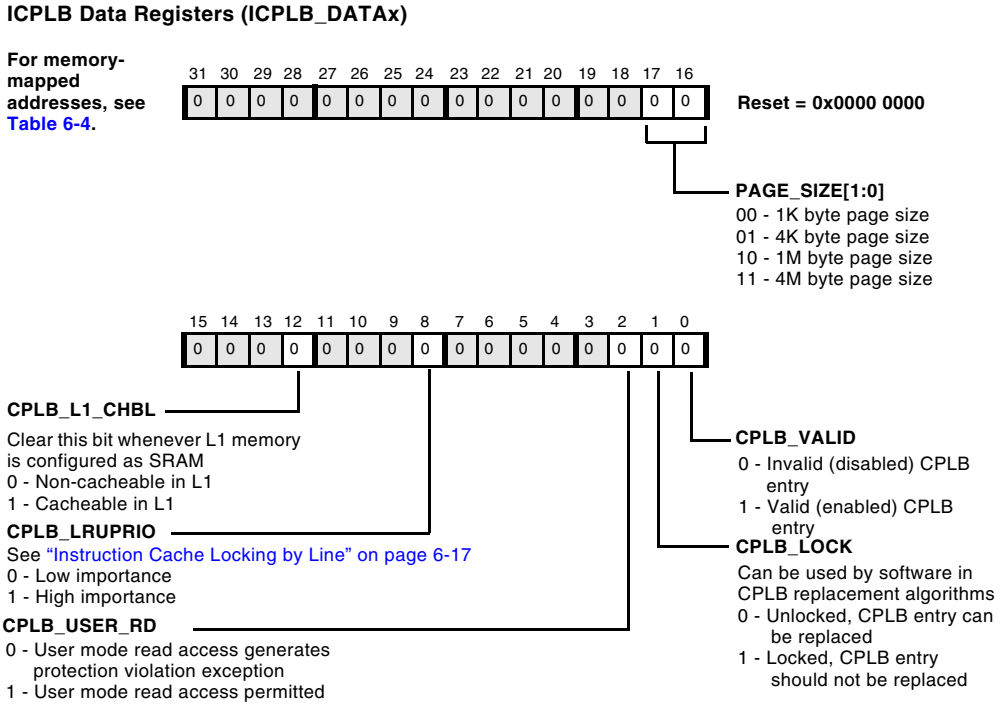


Figure 6-18. ICPLB Data Registers

Table 6-4. ICPLB Data Register Memory-mapped Addresses

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

Data CPLB Data (DCPLB_DATAx) Registers

Figure 6-19 shows the DCPLB data registers (DCPLB_DATAx).

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

DCPLB Data Registers (DCPLB_DATAx)

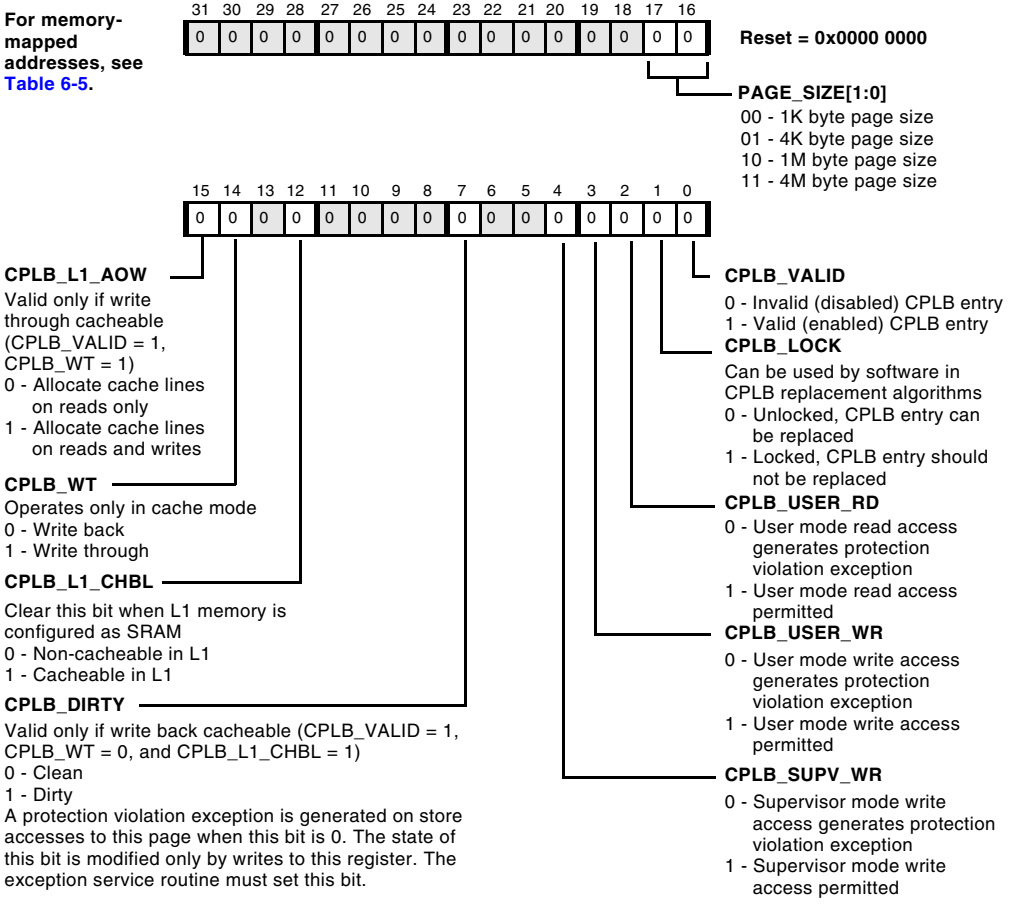


Figure 6-19. DCPLB Data Registers

Table 6-5. DCPLB Data Register Memory-Mapped Addresses

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

Data CPLB Address (DCPLB_ADDRx) Registers

Figure 6-20 shows the DCPLB address registers (DCPLB_ADDRx).

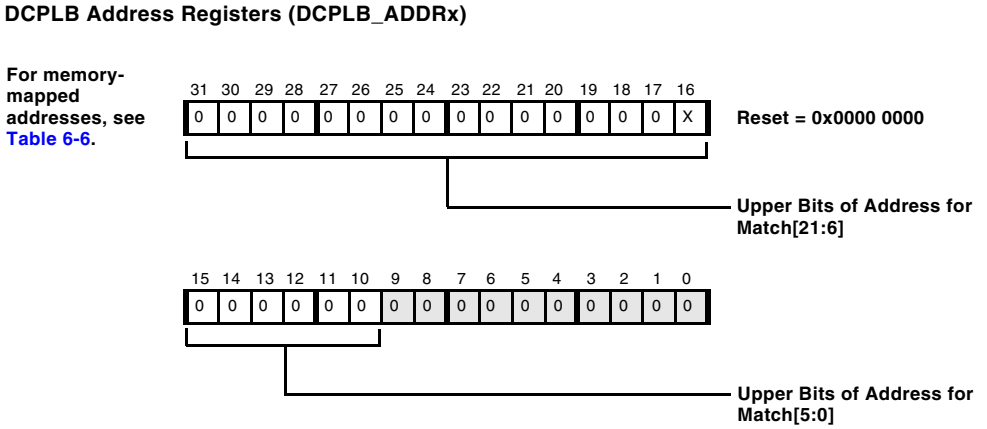


Figure 6-20. DCPLB Address Registers

Table 6-6. DCPLB Address Register Memory-Mapped Addresses

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

Instruction CPLB Address (ICPLB_ADDRx) Registers

Figure 6-21 shows the ICPLB Address registers (`ICPLB_ADDRx`).

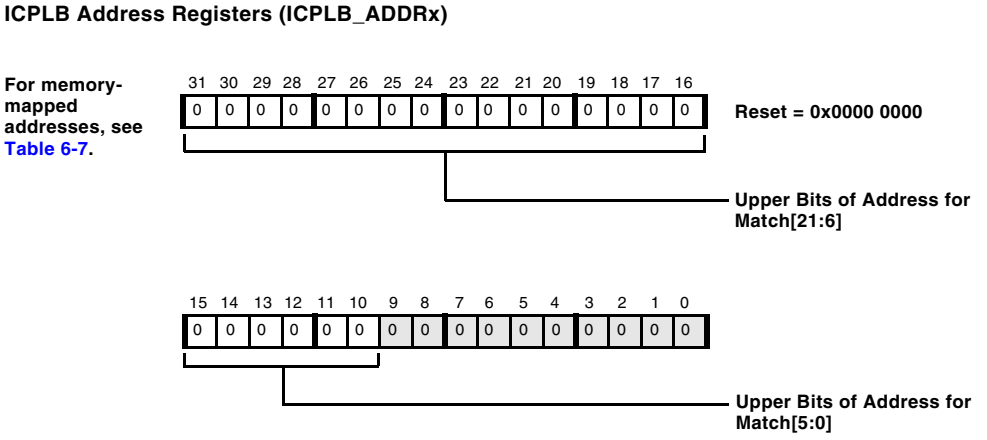


Figure 6-21. ICPLB Address Registers

Table 6-7. ICPLB Address Register Memory-mapped Addresses

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

Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers

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

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

Bits FAULT_DAG, FAULT_USERSUPV and FAULT_RW in the DCPLB status register (DCPLB_STATUS) are used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-22).

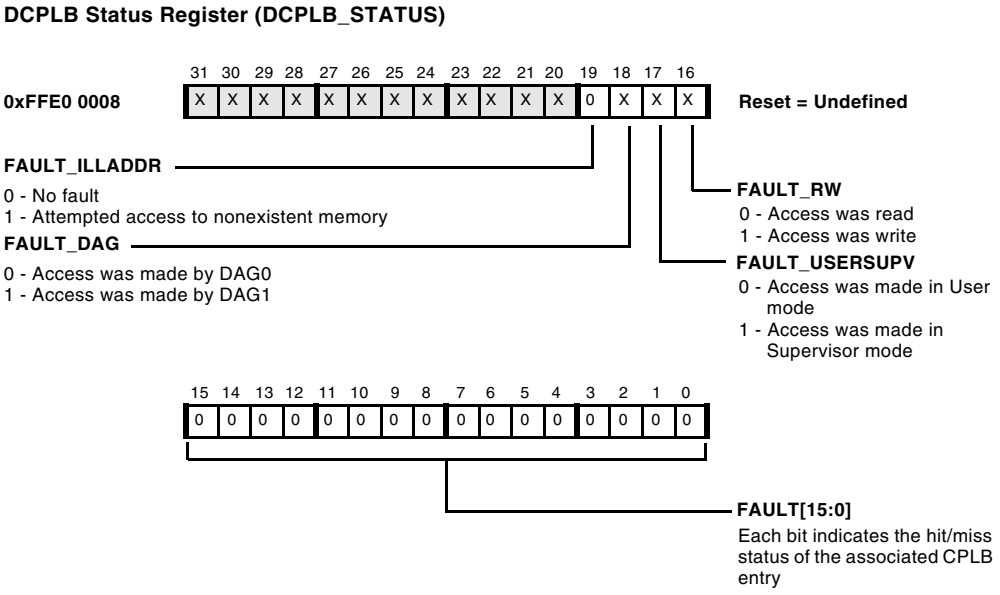


Figure 6-22. DCPLB Status Register

Bit FAULT_USERSUPV in the ICPLB status register (ICPLB_STATUS) is used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-23).

Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers

The DCPLB address register (DCPLB_FAULT_ADDR) and ICPLB Fault Address register (ICPLB_FAULT_ADDR) hold the address that has caused a fault in the L1 data memory or L1 instruction memory, respectively. See Figure 6-24 and Figure 6-25.

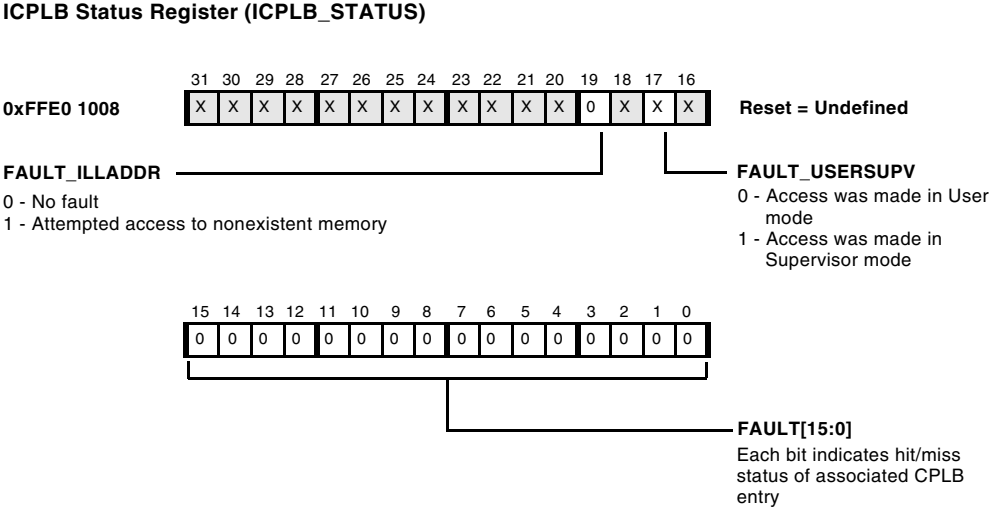


Figure 6-23. ICPLB Status Register

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

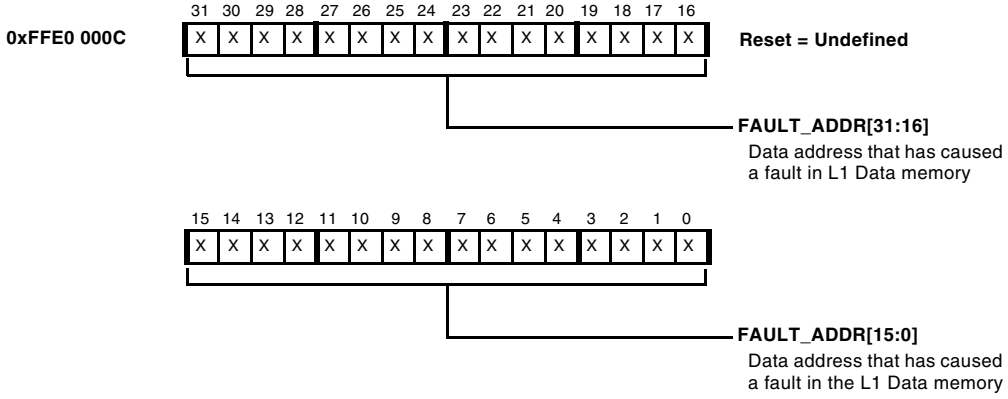


Figure 6-24. DCPLB Address Register

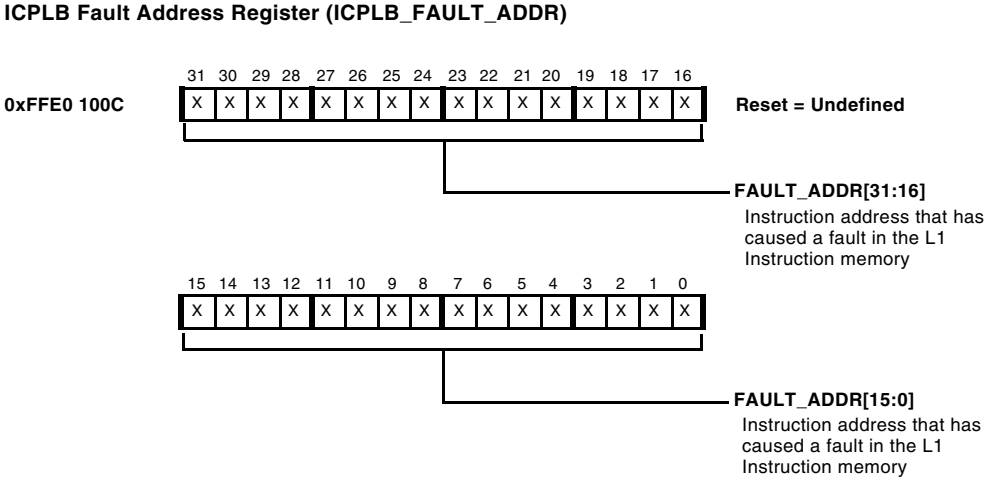


Figure 6-25. ICPLB Fault Address Register

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 6-26 shows a data word stored in register `R0` and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

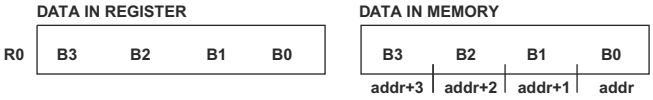


Figure 6-26. Data Stored in Little Endian Order

Figure 6-27 shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in *addr+1*) and the least significant byte in the low address (byte B0 in *addr*).

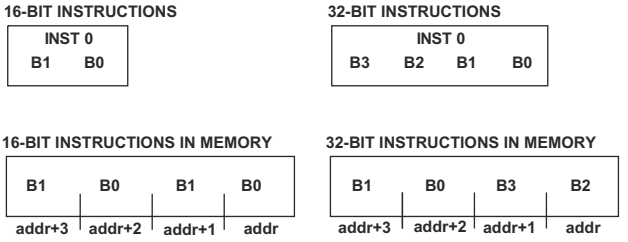


Figure 6-27. Instructions Stored in Little Endian Order

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

Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a Load/Store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

Separating load operations from their associated arithmetic functions allows compilers or assembly language programmers to place unrelated instructions between the load and its dependent instructions. If the value is returned before the dependent operation reaches the execution stage of the pipeline, the operation completes in one cycle.

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

Interlocked Pipeline

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

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

Ordering of Loads and Stores

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

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

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed. This ordering provides significant performance advantages in the operation of most memory instructions. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

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

Synchronizing Instructions

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

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

Consider the following example code sequence:

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

In the preceding example code, the CSYNC instruction ensures:

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

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

Speculative Load Execution

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

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

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

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

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

Conditional Load Behavior

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

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

Working With Memory

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

Alignment

Nonaligned memory operations are not directly supported. A nonaligned memory reference generates a Misaligned Access exception event (see "Exceptions" on page 4-45). However, because some datastreams (such as 8-bit video data) can properly be nonaligned in memory, alignment exceptions may be disabled by using the `DISALGNEXCPT` instruction. Moreover, some instructions in the quad 8-bit group automatically disable alignment exceptions.

Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the `FLUSH` instruction (see "Data Cache Control Instructions" on page 6-38), and/or explicit line invalidation through the core MMRs (see "Data Test Registers" on page 6-39).

Atomic Operations

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

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

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

Memory-mapped Registers

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

If strong ordering is required, place a synchronization instruction after stores to MMRs. For more information, see "Load/Store Operation" on page 6-63.

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

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

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

Appendix A provides a summary of all Core MMRs. Appendix B provides a summary of all System MMRs.

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Nonaligned access to MMRs generates an exception event. Listing 6-1 shows the instructions required to manipulate a generic core MMR.

Listing 6-1. Core MMR Programming

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

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

Terminology

The following terminology is used to describe memory.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

set. A group of *N*-line storage locations in the Ways of an *N*-Way cache, selected by the `INDEX` field of the address (see Figure 6-5 on page -13).

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

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

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

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

Way. An array of line storage elements in an *N*-Way cache (see Figure 6-5 on page -13).

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

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

7 CHIP BUS HIERARCHY

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

Internal Interfaces

Figure 7-1 on page 7-2 shows the core processor and system boundaries and the interfaces between them.

Internal Clocks

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

The Peripheral Access Bus (PAB), the DMA Access Buses (DAB0/DAB1), the External Access Bus (EAB), the DMA Core Buses (DCB0–2), the DMA External Bus (DEB), the External Port Bus (EPB), and the External Bus Interface Unit (EBIU) run at the system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL Divide register and must be set so that these buses run as specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*, and slower than or equal to the core clock frequency.

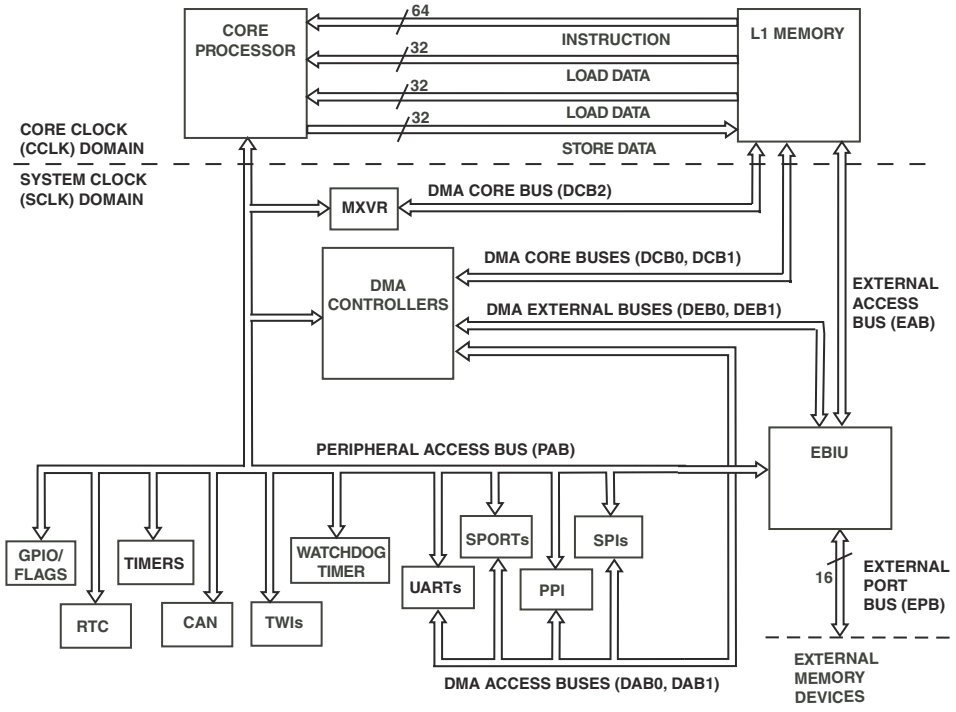


Figure 7-1. Processor Bus Hierarchy

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

Core Overview

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

The following block diagram shows the core processor and its interfaces to the peripherals and external memory resources.

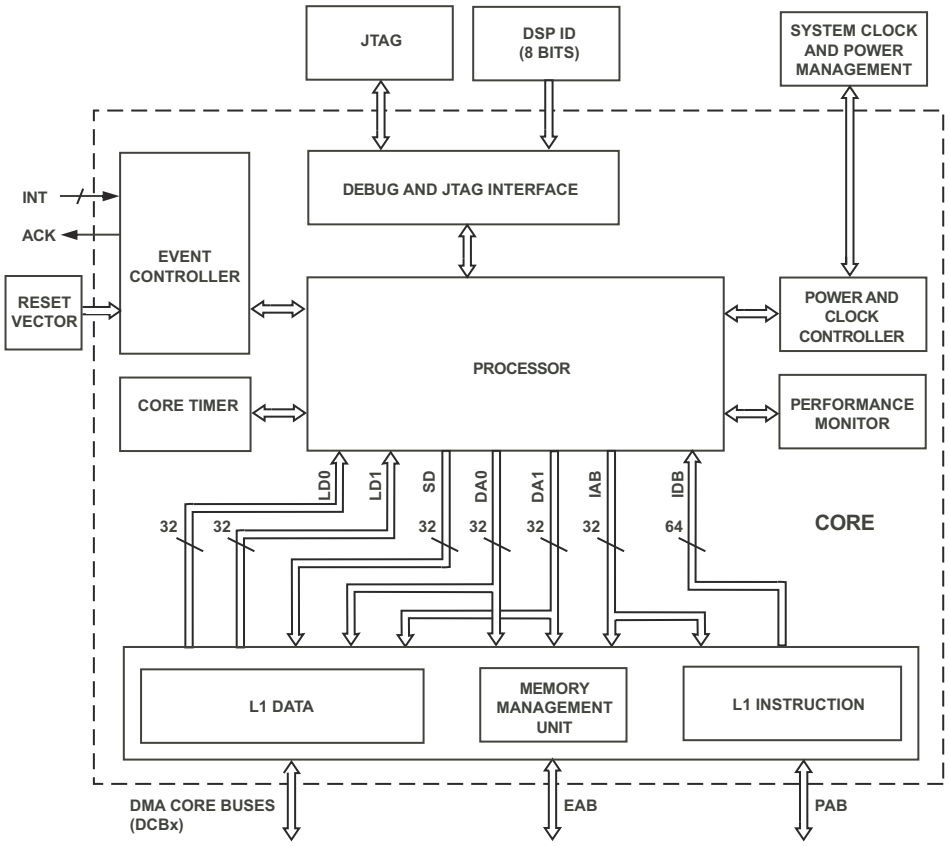


Figure 7-2. Core Block Diagram

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

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

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

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

System Overview

The system includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock-domain conversion is provided to support clock domain transactions between the core and the system.

System Interfaces

The processor system includes the peripheral set (timers, real time clock, programmable flags, general-purpose I/O, UARTs, SPORTs, PPI, watchdog timer, SPIs, TWIs, CAN, and MXVR), the external memory controller (EBIU), the DMA controllers, and the interfaces between these, the system, and the optional external (off-chip) resources. See Figure 7-2 on page 7-3.

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

- “Peripheral Access Bus (PAB)” on page 7-5
- “DMA Access Buses (DAB0/DAB1), DMA Core Buses (DCB0/DCB1/DCB2), DMA External Buses (DEB0/DEB1)” on page 7-7
- “External Access Bus (EAB)” on page 7-10

The External Bus Interface Unit (EBIU) is the primary chip pin bus. See Chapter 18, “External Bus Interface Unit”.

Peripheral Access Bus (PAB)

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

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte access to this region is not supported. Also, the TESTSET instruction to system memory mapped register space is not supported, since TESTSET produces byte-size read-modify-write.

PAB Arbitration

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

PAB Performance

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are 2 SCLK cycles.

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2 times the frequency of the system clock, the first and subsequent system MMR read or write accesses take 4 core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

PAB Agents (Masters, Slaves)

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

The slaves on the PAB bus are as follows:

- Event controller
- Clock and power management controller
- Watchdog timer
- Real time clock
- Timers0–2
- SPORT0–3
- SPI0–2
- Programmable Flags
- General Purpose I/O
- UART0–2
- PPI
- MXVR
- TWI0–1
- CAN
- Asynchronous memory controller (AMC)
- SDRAM controller (SDC)

- DMA controller 0
- DMA controller 1

DMA Access Buses (DAB0/DAB1), DMA Core Buses (DCB0/DCB1/DCB2), DMA External Buses (DEB0/DEB1)

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

DABx, DCBx, and DEBx Arbitration

There are 13 DMA capable peripherals in the processor system, including both memory DMA controllers. 26 DMA channels and bus masters support these devices. The peripheral DMA controllers can transfer data between peripherals and internal or external memory.

The DAB buses are implemented as two separate bus systems, each interfacing to a DMA controller and a fixed set of peripheral DMA bus masters. Each of the two DMA controllers, as well as the MXVR transceiver, access L1 memory through the DCB buses. In the event of simultaneous requests to L1 memory, access is granted to MXVR first, followed by DMA controller 0, with DMA controller 1 having the lowest priority. Each of the two DMA controllers access external memory through the DEB buses. In the event of simultaneous requests to external memory, access is granted to DMA controller 0 first. This fixed priority arrangement should be considered as each of the application specific interfaces are assigned to each peripheral.

Table 7-1 and Table 7-2 show the default arbitration priority of each DAB bus.

Table 7-1. Controller 0 (DAB0) Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
PPI	0 - highest
SPORT0 RX DMA controller	1
SPORT1 RX DMA controller	3
SPORT0 TX DMA controller	2
SPORT1 TX DMA controller	4
SPI0 DMA controller	5
UART0 RX controller	6
UART0 TX controller	7
Memory DMA0 (dest) controller	8
Memory DMA0 (source) controller	9
Memory DMA1 (dest) controller	10
Memory DMA1 (source) controller	11 - lowest

Table 7-2. Controller 1 (DAB1) Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
SPORT2 RX DMA controller	0 - highest
SPORT2 TX DMA controller	1
SPORT3 RX DMA controller	2
SPORT3 TX DMA controller	3
Unassigned	4
Unassigned	5
SPI1 DMA controller	6
SPI2 DMA controller	7

Table 7-2. Controller 1 (DAB1) Arbitration Priority (Cont'd)

DAB, DCB, DEB Master	Default Arbitration Priority
UART1 RX controller	8
UART1 TX controller	9
UART2 RX controller	10
UART2 TX controller	11
Memory DMA2 (dest) controller	12
Memory DMA2 (source) controller	13
Memory DMA3 (dest) controller	14
Memory DMA3 (source) controller	15 - lowest

The DCB has priority over the core processor on arbitration into L1 configured as data SRAM, whereas the core processor has priority over the DCB on arbitration into L1 instruction SRAM. For off-chip memory, the core has priority over the DEB buses on the EAB bus.

DAB, DCB, and DEB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

The DCB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4 KB size for L1). If there is a conflict when accessing data memory, DMA is the highest priority requester, followed by the core. If the conflict occurs when accessing instruction memory, the core is the highest priority requester, followed by DMA.

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.

DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. These latencies caused by these stalls are in addition to any arbitration latencies.

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

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 7-1 on page 7-8. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

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

External Access Bus (EAB)

The EAB provides a way for the processor core and the memory DMA controller to directly access off-chip memory and high throughput memory-to-memory DMA transfers.

EAB Arbitration

Arbitration for use of the external port bus interface resources is required because of possible contention between the potential masters of this bus. A fixed-priority arbitration scheme is used.

EAB Performance

The EAB supports single word accesses of either 8-bit or 16-bit data types. The EAB operates at the same frequency as the PAB and the DAB, up to the maximum SCLK frequency specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

Memory DMA transfers typically result in repeated accesses to the same memory location. Because the memory DMA controller has the potential of simultaneously accessing on-chip and off-chip memory, considerable throughput can be achieved. The throughput rate for an on-chip/off-chip memory access is limited by the slower of the two accesses. An additional 1 to 2 cycles per burst access is inherent in the design.

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

Table 7-2 on page 7-8 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations.

Table 7-3. EAB Performance

Source	Destination	Approximate SCLKs For n Words (from start of DMA to interrupt at end)
16-bit SDRAM	L1 Data memory	n + 14
L1 Data memory	16-bit SDRAM	n + 11
16-bit async memory	L1 Data memory	n + 12
L1 Data memory	16-bit async memory	n + 9
16-bit SDRAM	16-bit SDRAM	10 + (17n/7)
16-bit async memory	16-bit async memory	10 + 2xn, where x is the number of wait states + setup/hold SCLK cycles (minimum x=2)
L1 Data memory	L1 Data memory	2n + 12

8 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the processor. This functionality includes:

- Clocking
- Phase locked loop (PLL)
- Dynamic power management controller
- Operating modes
- Voltage control

Clocking

The input clock into the processor, CLKIN, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip Phase Locked Loop (PLL) module. During normal operation, the user programs the PLL with a multiplication factor for CLKIN. The resulting, multiplied signal is the voltage controlled oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (CCLK).

A user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the Peripheral Access bus (PAB), DMA buses (DABx), External Address bus (EAB), and the External Bus Interface Unit (EBIU).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL Divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a coarse adjustment. For a fine adjustment, the PLL clock frequency can also be varied.

Phase Locked Loop and Clock Control

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the dynamic power management controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see Chapter , “Dynamic Power Management Controller”

PLL Overview

Subject to the maximum VCO frequency, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

Figure 8-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

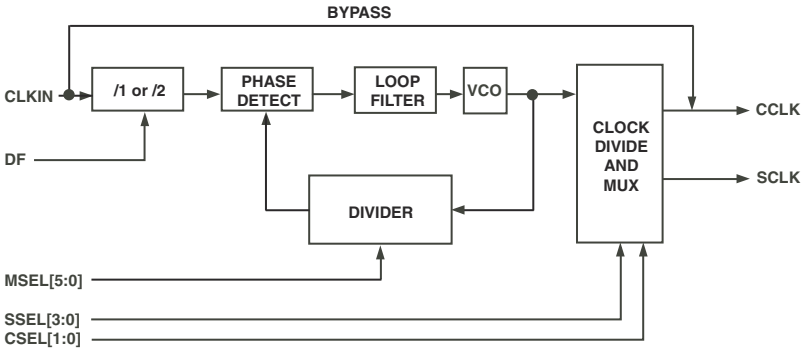


Figure 8-1. PLL Block Diagram

PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL Control (PLL_CTL) Register" on page 8-7

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers
  The reset value of MSEL is 0xA. This value can be reprogrammed at startup in the boot code.

Table 8-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 8-1. MSEL Encodings

Signal name MSEL[5:0]	VCO Frequency DF = 0	DF = 1
0	64x	32x
1	1x	0.5x
2	2x	1x
N = 3–62	Nx	0.5Nx
63	63x	31.5x

Core Clock/System Clock Ratio Control

Table 8-2 on page -5 describes the programmable relationship between the VCO frequency and the core clock. Table 8-3 on page -6 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to limit the SCLK to a frequency specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*. The SCLK drives all synchronous, system-level logic.

The divider ratio control bits, CSEL and SSEL, are in the PLL Divide register (PLL_DIV). For information about this register, see "PLL Divide (PLL_DIV) Register" on page 8-7. Appendix "System MMR Assignments" shows the register addresses.

The reset value of CSEL[1:0] is 0x0 (CCLK = VCO/1), and the reset value of SSEL[3:0] is 0x5 (SCLK = CCLK/5). These values can be reprogrammed at startup by the boot code.

By writing the appropriate value to PLL_DIV, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

The PLL_DIV register can be programmed at any time to change the CCLK and SCLK divide values without entering the Idle state.

Table 8-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
b#00	1	300	300
b#01	2	600	300
b#10	4	600	150
b#11	8	400	50

As long as the MSEL and DF control bits in the PLL control register (PLL_CTL) remain constant, the PLL is locked.

Table 8-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
b#0000	Reserved	N/A	N/A
b#0001	1:1	100	100
b#0010	2:1	200	100
b#0011	3:1	400	133
b#0100	4:1	500	125
b#0101	5:1	600	120
b#0110	6:1	600	100
N = 7–15	N:1	600	600/N

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

PLL Registers

The user interface to the PLL is through four memory-mapped registers (MMRs):

- The PLL divide register (PLL_DIV)
- The PLL control register (PLL_CTL)
- The PLL status register (PLL_STAT)
- The PLL lock count register (PLL_LOCKCNT)

All four registers are 16-bit MMRs and must be accessed with aligned 16-bit reads/writes.

PLL Divide (PLL_DIV) Register

The PLL divide register (PLL_DIV) divides the PLL output clock to create the processor core clock (CCLK) and the system clock (SCLK). These values can be independently changed during processing to reduce power dissipation without changing the PLL state. The only restrictions are the resulting CCLK frequency must be greater than or equal to the SCLK frequency, and SCLK must fall within the allowed range specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*. If the CCLK and SCLK divide values are programmed otherwise, the SCLK value is automatically adjusted to be slower than or equal to the core clock. Figure 8-2 shows the bits in the PLL_DIV register.

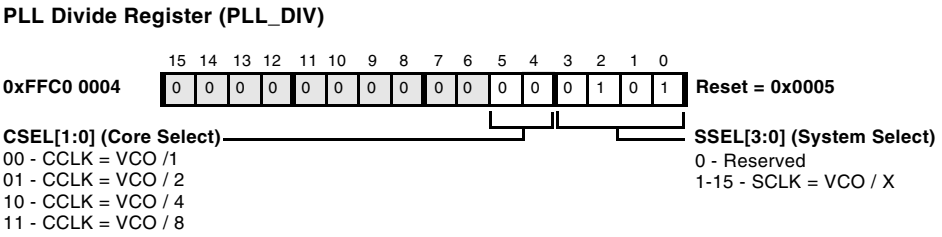


Figure 8-2. PLL Divide Register

PLL Control (PLL_CTL) Register

The PLL control register (PLL_CTL) controls operation of the PLL (see Figure 8-3). Note changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with new values, and then a specific PLL programming sequence must be executed to implement the changes. See "PLL Programming Sequence" on page 8-19

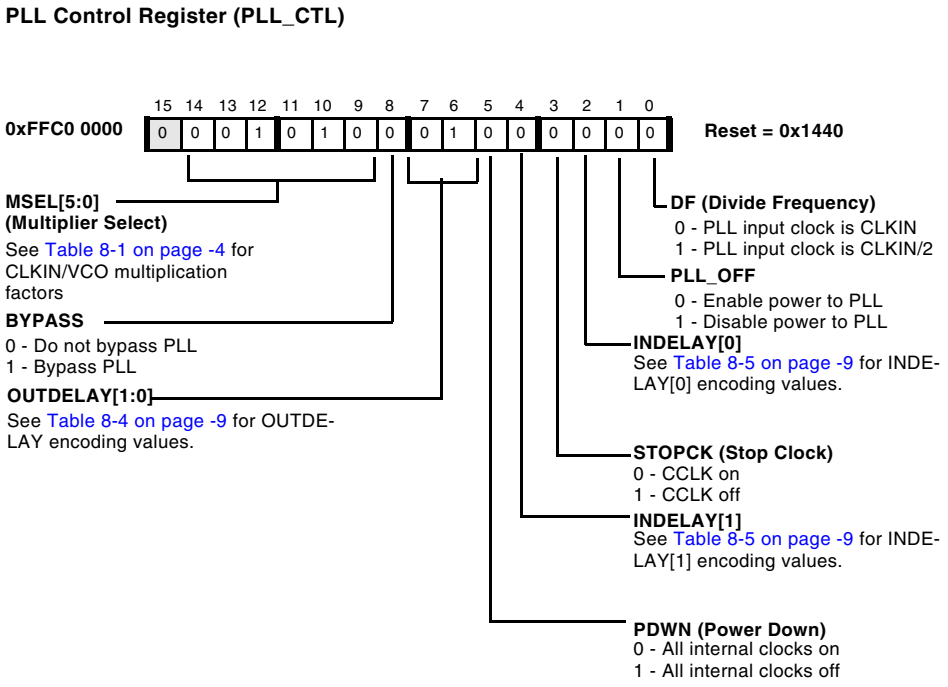


Figure 8-3. The PLL Control Register

The following fields of the PLL_CTL register are used to control the PLL:

- MSEL[5:0] – The multiplier select (MSEL) field defines the PLL input clock to VCO clock multiplier. Changes to this field are permitted only when in Full On mode.
- BYPASS – This bit is used to bypass the PLL. When BYPASS is set, CLKIN is passed directly to the core and peripheral clocks.
- OUTDELAY[1:0] – These bits are used to add 200 ps or 400 ps of delay or subtract 200 ps of delay from the output delay of the external memory signals. The reset value of the OUTDELAY[1:0] is b#01 and all writes to this register should be careful to maintain the contents of this field when programming other fields.

Table 8-4. OUTDELAY Encodings

Encoding	Description
b#00	Do not add output delay
b#01	Add approximately 400 ps (default)
b#10	Subtract approximately 200 ps
b#11	Subtract approximately 400 ps

- `INDELAY[1:0]` – These bits are used to subtract 200 ps or 400 ps of delay from or add 200 ps of delay to the time when inputs are latched on the external memory interface. The reset value of the `INDELAY[1:0]` is b#00 and all writes to this register should be careful to should be careful to maintain the contents of this field when programming other fields.

Table 8-5. INDELAY Encodings

Encoding	Description
b#00	Do not add input delay (default)
b#01	Add approximately 200 ps
b#10	Subtract approximately 200 ps
b#11	Subtract approximately 400 ps

- PDWN – The power down (PDWN) bit is used to place the processor in the Deep Sleep operating mode.

  For information about operating modes, see "Operating Modes" on page 8-13

- STOPCK – The stop clock (STOPCK) bit is used to enable/disable the core clock, CCLK.

- PLL_OFF – This bit is used to enable/disable power to the PLL.

- DF – The divide frequency (DF) bit controls the PLL input clock divider which determines whether the PLL input clock is CLKIN or CLKIN/2.

PLL Status (PLL_STAT) Register

The PLL status register (PLL_STAT) indicates the operating mode of the PLL and processor (see Figure 8-4). For more information about operating modes, see "Operating Modes" on page 8-13.

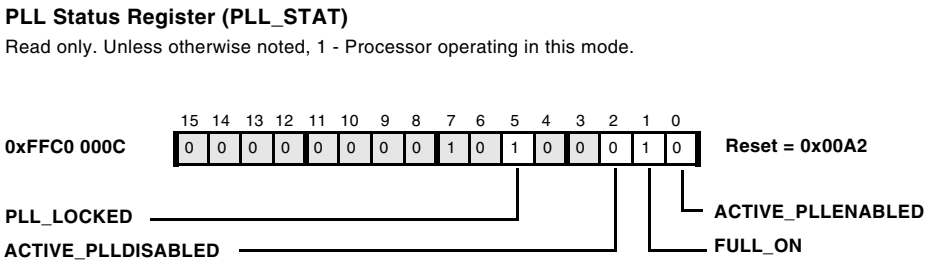


Figure 8-4. PLL Status Register

The following fields are used in the PLL_STAT register:

- PLL_LOCKED – This field is set to 1 when the internal PLL lock counter has incremented to the value set in the PLL lock count register (PLL_LOCKCNT). For more information, see "PLL Lock Count (PLL_LOCKCNT) Register" on page 8-11.
- ACTIVE_PLLDISABLED – This field is set to 1 when the processor is in active operating mode with the PLL powered down.
- FULL_ON – This field is set to 1 when the processor is in full on operating mode.
- ACTIVE_PLLENABLED – This field is set to 1 when the processor is in active operating mode with the PLL powered up.

PLL Lock Count (PLL_LOCKCNT) Register

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency.

The PLL lock count register (PLL_LOCKCNT) defines the number of SCLK cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each SCLK cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

The PLL lock counter will begin incrementing after the PLL is bypassed for the PLL relock sequence. Therefore, though it is clocked by SCLK, the rate at which the count increments is CLKIN.

See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see "Operating Modes" on page 8-13. For further information about the PLL programming sequence, see "PLL Programming Sequence" on page 8-19.

PLL Lock Count Register (PLL_LOCKCNT)

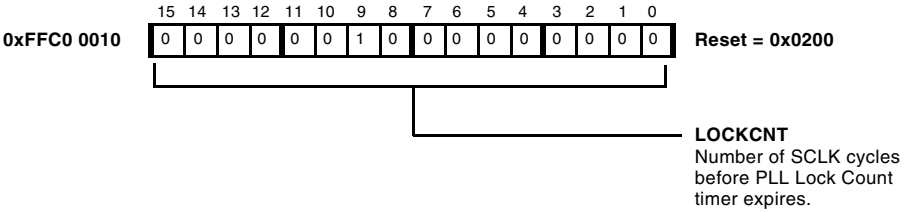


Figure 8-5. PLL Lock Count Register

Dynamic Power Management Controller

The dynamic power management controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor's performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See "Operating Modes" on page 8-13.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.
- Voltage control – The processor provides an on-chip switching regulator controller which, with some external components, can generate internal voltage levels from an external (V_{DDEXT}) supply.

Depending on the needs of the system, the voltage level can be reduced to save power. See "Voltage Regulator Control (VR_CTL) Register" on page 8-26.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 8-6 summarizes the operational characteristics of each mode.

Table 8-6. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The state of the DPMC/PLL can be determined by reading the PLL status register (see "PLL Status (PLL_STAT) Register" on page 8-10). In all modes except Sleep and Deep Sleep, the core can either execute instructions or be in Idle core state. If the core is in the Idle state, it can be awakened.

In all modes except Active, the SCLK frequency is determined by the SSEL-specified ratio to VCO. In Sleep mode, although the core clock is disabled, SCLK continues to run at the specified SSEL ratio.

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full On Mode

Full On mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full On mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. DMA access is available to L1 memories. From Full On mode, the processor can transition directly to Active, Sleep, or Deep Sleep modes, as shown in Figure 8-6 on page 8-17.

Active Mode

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

In Active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to Full On or Sleep modes.

From Active mode, the processor can transition directly to Full On, Sleep, or Deep Sleep modes.

Sleep Mode

Sleep mode significantly reduces dynamic power dissipation by idling the core processor. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. As CCLK is disabled, DMA access is available only to external memory in Sleep mode. From Sleep mode, a wakeup event enabled in the SIC_IWRx registers causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full On mode if the BYPASS bit is cleared

The processor resumes execution from the program counter value present immediately prior to entering sleep mode.

The STOPCK bit is not a status bit and is therefore unmodified by hardware when the wakeup occurs. Software must explicitly clear STOPCK in the next write to PLL_CTL to avoid going back into sleep mode.

Deep Sleep Mode

Deep Sleep mode maximizes dynamic power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals except the Real-Time Clock (RTC) are disabled. DMA is not supported in this mode.

Deep Sleep mode can be exited only by an RTC interrupt or hardware reset event. An RTC interrupt causes the processor to transition to Active mode; a hardware reset begins the hardware reset sequence. For more information about hardware reset, see "Hardware Reset" on page 3-14.

Note an RTC interrupt in Deep Sleep mode automatically resets some fields of the PLL control register (PLL_CTL). See Table 8-7.

When in Deep Sleep operating mode, clocking to the SDRAM is turned off. Before entering Deep Sleep mode, software should either ensure that important information in SDRAM is saved to a non-volatile memory or place the SDRAM into self-refresh mode.

Table 8-7. Control Register Values after RTC Wakeup Interrupt

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 8-6 on page 8-17. Since this feature is coupled to the on-chip switching regulator controller, it is discussed in detail in "Powering Down the Core (Hibernate State)" on page 8-31.

Operating Mode Transitions

Figure 8-6 on page 8-17, graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes. Arrows between the ellipses show the allowed transitions into and out of each mode.

The text next to each transition arrow shows the fields in the PLL control register (`PLL_CTL`) that must be changed for the transition to occur. For example, the transition from Full On mode to Sleep mode indicates that the `STOPCK` bit must be set to 1 and the `PDWN` bit must be set to 0. For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 8-19.

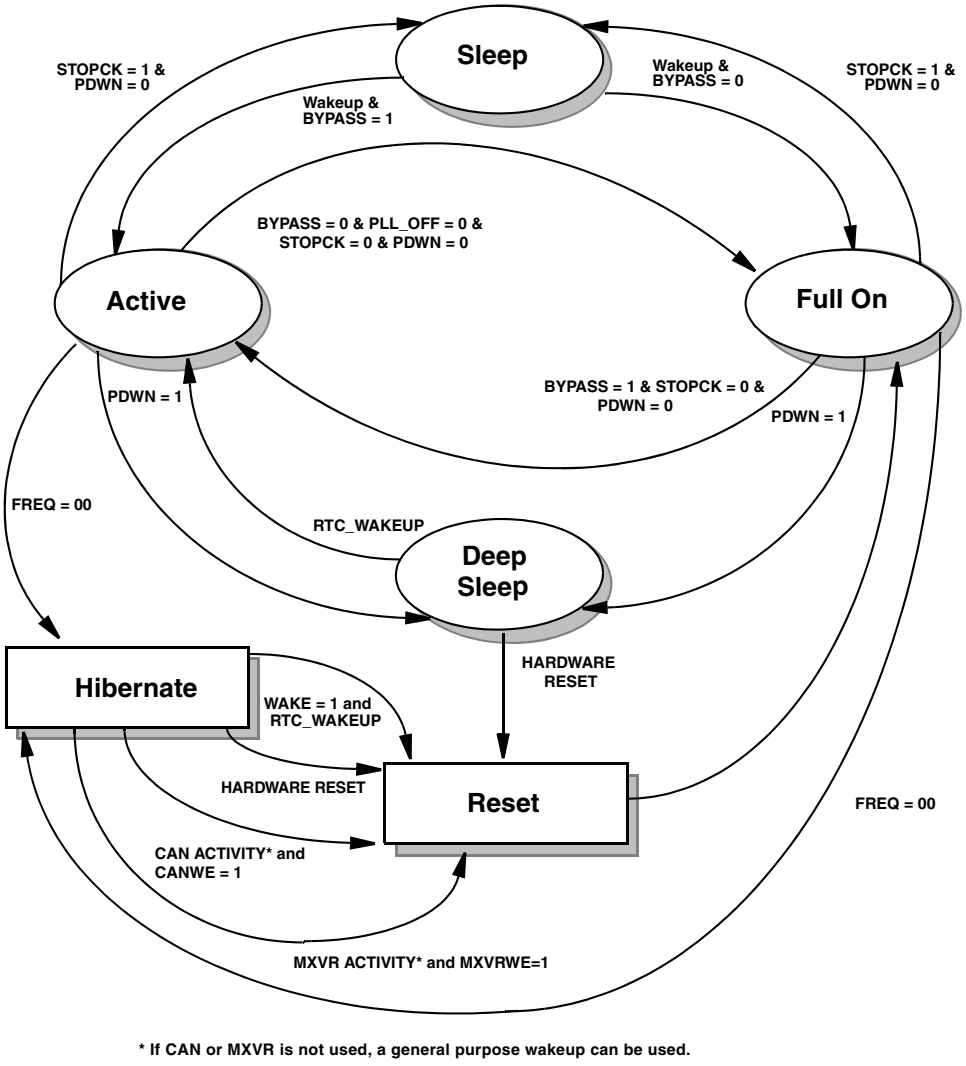


Figure 8-6. Operating Mode Transitions

- PLL Disabled: In addition to being bypassed in the Active mode, power to the PLL can be removed. When power is removed from the PLL, additional power savings are achieved although they are relatively small. To remove power to the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.
- PLL enabled: When the PLL is powered down, power can be reapplied later when additional performance is required.

  Power to the PLL must be reapplied before transitioning to Full On or Sleep operating modes. To apply power to the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.
- New Multiplier Ratio: The multiplier ratio can be changed while in Full On mode.

  The PLL state automatically transitions to Active mode while the PLL is locking. After locking, the PLL returns to Full On state. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence (see "PLL Programming Sequence" on page 8-19).

Table 8-8 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 8-8 causes unpredictable behavior.

Table 8-8. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	–
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control register (PLL_CTL). Merely modifying the bits of the PLL_CTL register does not change the operating mode or the behavior of the PLL. Changes to the PLL_CTL register are realized only after executing a specific code sequence, which is shown in Listing 8-1. This code sequence first brings the processor to a known, idled state. Once in this idled state, the PLL recognizes and implements the changes made to the PLL_CTL register. After the changes take effect, the processor operates with the new settings, including the new operating mode, if one is programmed.

PLL Programming Sequence

If new values are assigned to MSEL or DF in the PLL control register (PLL_CTL), the instruction sequence shown in Listing 8-1 puts those changes into effect. The PLL programming sequence is also executed when transitioning between operating states.

Changes to the divider-ratio bits, CSEL and SSEL, can be made dynamically; they do not require execution of the PLL programming sequence.

Listing 8-1. PLL Programming Sequence

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

The first two instructions in the sequence take the core to an idled state with interrupts disabled; the interrupt mask (IMASK) is saved to the R0 register, and the instruction pipeline is halted. The PLL state machine then loads the PLL_CTL register changes into the PLL. In order to break from the idled state, the PLL wakeup event must be enabled in the system interrupt controller interrupt wakeup register (set bit 0 of SIC_IWR0).

To absolutely protect against asynchronous events compromising this sequence, user code can clear all other wake-up events in the SIC_IWRx registers and perform the CLI instruction before the write to the PLL_CTL register.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or the changes reapply power to the PLL, the PLL needs to re-lock. To re-lock, the PLL lock counter is first cleared, and then it begins incrementing, once per SCLK cycle. After the PLL lock counter reaches the value programmed into the PLL Lock Count register (PLL_LOCKCNT), the PLL sets the PLL_LOCKED bit in the PLL status register (PLL_STAT), and the PLL asserts the PLL wakeup interrupt.

If the new value written to the PLL_CTL or VR_CTL register is the same as the previous value, the PLL wake-up will occur immediately (PLL is already locked), but the core and system clock will be bypassed for the PLL_LOCKCNT duration. For this interval, code will

execute at the CLKIN rate instead of at the expected CCLK rate. Software should guard against this condition by comparing the current value to the new value before writing the new value.

Depending on how the PLL_CTL register is programmed, the processor proceeds in one of the following four ways:

- If the PLL_CTL register is programmed to enter either Active or Full On operating mode, the PLL generates a wakeup signal, and then the processor continues with the STI instruction in the sequence, as described in "PLL Programming Sequence Continues" on page 8-22.

  When the state change enters Full On mode from Active mode or Active from Full On, the PLL itself generates a wakeup signal that can be used to exit the idled core state. The wakeup signal is generated by the PLL itself or another peripheral, watchdog or other timer, RTC, or other source. For more information about events that cause the processor to wakeup from being idled, see "System Interrupt Wakeup-Enable (SIC_IWRx) Registers" on page 4-26.

- If the PLL_CTL register is programmed to enter the Sleep operating mode, the processor immediately transitions to the Sleep mode and waits for a wakeup signal (enabled in SIC_IWRx) before continuing.

  When the wakeup signal has been asserted, the instruction sequence continues with the STI instruction, as described in the section, "PLL Programming Sequence Continues," causing the processor to transition to:

  - —Active mode if BYPASS in the PLL_CTL register is set

    —Full On mode if the BYPASS bit is cleared

- If the PLL_CTL register is programmed to enter Deep Sleep operating mode, the processor immediately transitions to Deep Sleep mode and waits for an RTC interrupt or hardware reset signal:
    - —An RTC interrupt causes the processor to enter Active operating mode and continue with the STI instruction in the sequence, as described below.

      —A hardware reset causes the processor to execute the reset sequence, as described in "Reset" on page 4-44.
- If no operating mode transition is programmed, the PLL generates a wakeup signal, and the processor continues with the STI instruction in the sequence, as described in the following section.

PLL Programming Sequence Continues

The instruction sequence shown in Listing 8-1 on page -20 then continues with the STI instruction. Interrupts are re-enabled, IMASK is restored, and normal program flow resumes.

To prevent spurious activity, DMA should be suspended while executing this instruction sequence.

Examples

The following code examples illustrate how to effect various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- P0 points to the PLL control register (PLL_CTL). P1 points to the PLL Divide register (PLL_DIV).
- The PLL wakeup interrupt is enabled as a wakeup signal.
- MSEL[5:0] and DF in PLL_CTL are set to (b#011111) and (b#0) respectively, signifying a CLKIN to VCO multiplier of 31x. OUTDELAY[1:0] is set to (b#01) and INDELAY[1:0] is set to (b#00).

Active Mode to Full On Mode

Listing 8-2 provides code for transitioning from Active operating mode to Full On mode.

Listing 8-2. Transitioning From Active Mode to Full On Mode

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3E40;   /* clear BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Full On mode */
```

Full On Mode to Active Mode

Listing 8-3 provides code for transitioning from Full On operating mode to Active mode.

Listing 8-3. Transitioning From Full On Mode to Active Mode

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3F40;   /* set BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Active mode */
```

In the Full On Mode, Change CLKIN to VCO Multiplier From 31x to 2x

Listing 8-4 provides code for changing CLKIN to VCO multiplier from 31x to 2x in Full On operating mode.

Listing 8-4. Changing CLKIN to VCO Multiplier

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x0440;   /* change VCO multiplier to 2x */
W[P0] = R1;   /* by writing to PLL_CTL */

IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Full On mode, with the CLKIN to VCO
multiplier set to 2x */
```

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core processor at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses five power domains. These power domains are shown in Table 8-9. Each power domain has a separate V_{DD} supply. Note the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for details on the allowed voltage ranges for each power domain and power dissipation data.

Table 8-9. Power Domains

Power Domain	VDD Range
RTC Crystal I/O and Logic	VDDRTC
MXVR Crystal I/O	MXEVDD
MXVR PLL Analog and Logic	MPIVDD
All Internal Logic Except RTC and MXVR PLL	VDDINT
All I/O Except RTC and MXVR Crystals	VDDEXT

Power Supply Management

The processor provides an on-chip switching regulator controller which, with some external hardware, can generate internal voltage levels from the external V_{DDEXT} supply with an external power transistor as shown in Figure 8-7. This voltage level can be reduced to save power, depending upon the needs of the system.

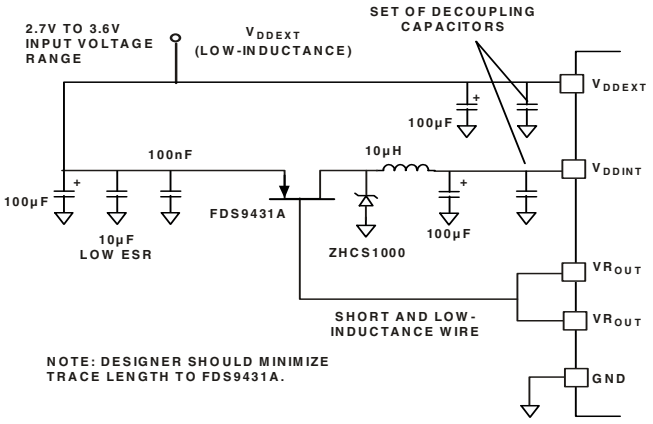


Figure 8-7. Processor Voltage Regulator

When increasing the V_{DDINT} voltage, the external FET will switch on for a longer period. The V_{DDEXT} supply should have appropriate capacitive bypassing to enable it to provide sufficient current without drooping the supply voltage.

Voltage Regulator Control (VR_CTL) Register

The on-chip core voltage regulator controller manages the internal logic voltage levels for the V_{DDINT} supply. The voltage regulator control register (VR_CTL) controls the regulator (see Figure 8-8). Writing to VR_CTL initiates a PLL re-lock sequence.

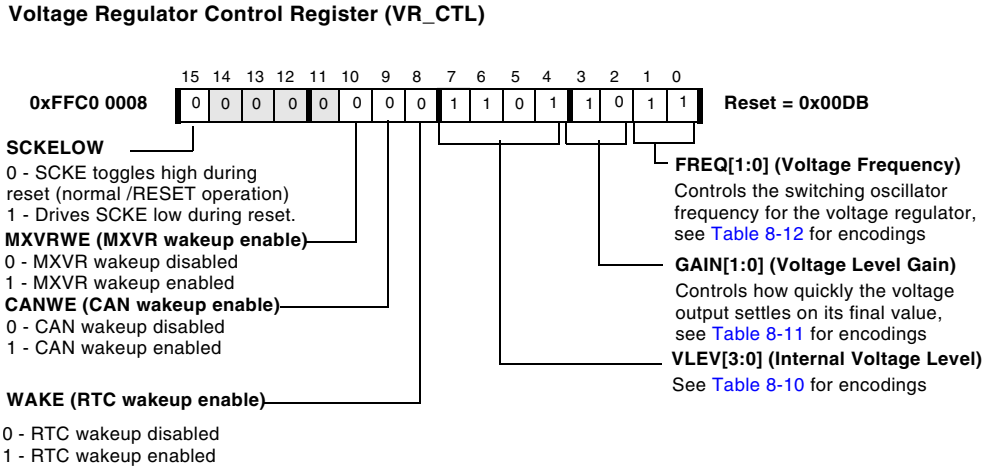


Figure 8-8. Voltage Regulator Control Register

The following fields of the VR_CTL register are used to control internal logic voltage levels:

- SCKELOW – The drive SCKE low during reset (SCKELOW) control bit protects against the default reset state behavior of setting the EBIU pins to their inactive state. This bit should be set if the SDRAM has been properly configured and is being placed into self-refresh mode while the processor is in hibernate state. Failure to set this bit results in the SCKE pin going high during reset, which takes the SDRAM out of self-refresh mode, resulting in data decay in the SDRAM due to loss of refresh rate. The SCKELOW bit maintains its state through hibernate and can be interrogated upon start-up to determine whether the processor is returning from hibernate or being cold-started.

- CANWE – The wakeup-enable (CANWE) control bit allows the voltage regulator to be awakened from hibernate (FREQ = b#00) when a falling edge on the CANRX input pin is detected. If CAN is not used, a general-purpose wake-up can be used instead, where the processor waits for any falling edge on this pin.

- MXVRWE – The wakeup-enable (MXVRWE) control bit allows the voltage regulator to be awakened from hibernate (FREQ = b#00) when a falling edge on the $\overline{\text{MRXON}}$ input pin is detected. If MXVR is not used, a general-purpose wake-up can be used instead, where the processor waits for any falling edge on this pin.

The $\overline{\text{MRXON}}$ pin is a 5 V tolerant input-only pin.

- WAKE – The wakeup-enable (WAKE) control bit allows the voltage regulator to be awakened from hibernate (FREQ = b#00) upon an interrupt from the RTC. This can be any of the RTC interrupt sources (alarm, day alarm, day, hour, minute, second, or stopwatch).

- VLEV[3:0] – The voltage level (VLEV) field identifies the nominal internal voltage level. Refer to *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for the applicable VLEV voltage range and associated voltage tolerances.

- FREQ[1:0] – The frequency (FREQ) field controls the switching oscillator frequency for the voltage regulator. A higher frequency setting allows for smaller switching capacitor and inductor values, while potentially generating more EMI (electromagnetic interference).

To bypass onboard regulation, program the FREQ field to b#00, WAKE to b#0, CANWE to b#0, MXVRWE to b#0, and leave the VROUT pins floating.

- GAIN[1:0] – The gain (GAIN) field controls the internal loop gain of the switching regulator loop; this bit controls how quickly the voltage output settles on its final value. In general, higher gain allows for quicker settling times but causes more overshoot in the process.

Table 8-10 lists the voltage level values for VLEV[3:0].

Refer to *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for details regarding acceptable VLEV values for the processor used.

Table 8-10. VLEV Encodings

VLEV	Voltage
b#0000–0101	Reserved
b#0110	.85 volts
b#0111	.90 volts
b#1000	.95 volts
b#1001	1.00 volts
b#1010	1.05 volts
b#1011	1.10 volts
b#1100	1.15 volts
b#1101	1.20 volts
b#1110	1.25 volts
b#1111	Reserved

Table 8-11 lists the switching frequency values configured by `FREQ[1:0]`.

Table 8-11. FREQ Encodings

FREQ	Value
b#00	Powerdown/Bypass onboard regulation
b#01	333 kHz
b#10	667 kHz
b#11	1MHz

Table 8-12 lists the gain levels configured by GAIN[1:0].

Table 8-12. GAIN Encodings

GAIN	Value
b#00	5
b#01	10
b#10	20
b#11	50

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for more information about supported voltage levels, regulator tolerances, and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior when varying the operating voltage, the processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. After changing the voltage level in the VR_CTL register, the PLL will automatically enter the Active mode when the processor enters the Idle state. At that point the voltage level will change and the PLL will re-lock with the new voltage. After the PLL_LOCKCNT has expired, the part will return to the full-on state. When changing voltages, a larger PLL_LOCKCNT value may be necessary than when changing just the PLL frequency. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for details.

After the voltage has been changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for the new operating voltage level.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be put into Hibernate State by writing the FREQ bits to b#00 and setting at least one of the wake up enables (WAKE, CANWE, MXVRWE) to 1 in the VR_CTL register. This disables both CCLK and SCLK. Furthermore, it sets the internal power supply voltage (V_{DDINT}) to 0 V, eliminating any leakage currents from the processor. The internal supply regulator can be woken up by several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the $\overline{\text{RESET}}$ pin will always exit hibernate state and requires no modification to VR_CTL.
- RTC event. Set the wakeup-enable (WAKE) control bit to enable wakeup upon a RTC interrupt. This can be any of the RTC interrupts (alarm, daily alarm, day, hour, minute, second, or stopwatch).
- MXVR event. Set the MXVR wakeup enable (MXVRWE) control bit to enable wakeup upon detection of MXVR activity (sense an active low signal) on the $\overline{\text{MRXON}}$ pin. See Chapter 21, “Media Transceiver Module (MXVR)”.
- CAN event. Set the CAN RX wakeup enable (CANWE) control bit to enable wakeup upon detection of CAN bus activity (sense an active low signal) on the CANRX pin. See Chapter 19, “CAN Module”.

For the peripheral hibernate wakeup sources described above, a general-purpose wakeup can be implemented if the peripheral is not used. For example, if MXVR is not used, an external host can be connected to the $\overline{\text{MRXON}}$ pin that holds the pin high until the wake-up is required. If MXVRWE is set, a transition to low on $\overline{\text{MRXON}}$ will exit hibernate, and the host could be set up to provide this signal. The same can be done if the host drives the CANRX pin and CANWE is set.

If the on-chip supply controller is bypassed, so that V_{DDINT} is sourced externally, the only way to power down the core is to remove the external V_{DDINT} voltage source.

When the core is powered down, V_{DDINT} is set to 0 V, and thus the internal state of the processor is not maintained. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a three-state level, unless otherwise specified.

To enter Hibernate State:

1. Write 0 to all bits in the SIC_IWRx registers to prevent enabled peripheral resources from interrupting the hibernate process
2. Write to VR_CTL, setting the FREQ bits to b#00 and enable any of the desired wake-up sources by setting the appropriate bits to 1 (WAKE, CANWE, and/or MXVRWE). Optionally, set the SCKELOW bit if SDRAM data should be maintained.
3. Execute the PLL reprogramming sequence.

4. When the Idle state is reached, V_{DDINT} will transition to 0 V.
5. When the processor is woken up, the PLL relocks and the boot sequence defined by the `BMODE[1:0]` pin settings takes effect.

9 DIRECT MEMORY ACCESS

The processor uses direct memory access (DMA) to transfer data within memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controllers carry out the data transfers independent of processor activity.

The processor contains three DMA engines. DMA controller 0 and DMA controller 1 are two instances of the DMA engine documented in this chapter. In addition to these two DMA controllers, the processor has a third DMA engine dedicated to MXVR. For more information see Chapter 21, “Media Transceiver Module (MXVR)”.

The DMA controllers can perform several types of data transfers:

- Between memory and memory DMA (MDMA), for more information see Chapter , “Memory DMA”.
- Between memory and a serial peripheral interface (SPI), for more information see Chapter 10, “SPI Compatible Port Controllers”.
- Between memory and a serial port (SPORT), for more information see Chapter 12, “Serial Port Controllers”.
- Between memory and a UART port, for more information see Chapter 13, “UART Port Controllers”.
- Between memory and the parallel peripheral interface (PPI), for more information see Chapter 11, “Parallel Peripheral Interface”.

The system includes 13 DMA-capable peripherals, including memory DMA controllers (MDMAx). The following 26 DMA channels support these devices:

Table 9-1. DMA Channels

PPI receive/transmit DMA controller
SPORT0 receive DMA controller
SPORT0 transmit DMA controller
SPORT1 receive DMA controller
SPORT1 transmit DMA controller
SPORT2 receive DMA controller
SPORT2 transmit DMA controller
SPORT3 receive DMA controller
SPORT3 transmit DMA controller
SPI0 receive/transmit DMA controller
SPI1 receive/transmit DMA controller
SPI2 receive/transmit DMA controller
UART0 receive DMA controller
UART0 transmit DMA controller
UART1 receive DMA controller
UART1 transmit DMA controller
UART2 receive DMA controller
UART2 transmit DMA controller
MDMA0 stream0 transmit (destination)
MDMA0 stream0 receive (source)
MDMA0 stream1 transmit (destination)

Table 9-1. DMA Channels (Cont'd)

MDMA0 stream1 receive (source)
MDMA1 stream0 transmit (destination)
MDMA1 stream0 receive (source)
MDMA1 stream1 transmit (destination)
MDMA1 stream1 receive (source)

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in "DAB, DCB, and DEB Performance" on page 7-9.

DMA transfers on the processor can be descriptor based or register based. Descriptor based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes. Register based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

DMA and Memory DMA MMRs

For convenience, discussions in this chapter use generic (non-peripheral-specific) DMA and memory DMA register names.

Generic DMA register names are listed in Table 9-2 on page 9-4.

Generic memory DMA register names are listed in Table 9-4 on page 9-9.

DMA registers fall into three categories:

- Current registers, such as `DMAx_CURR_ADDR` and `DMAx_CURR_X_COUNT`.
- Parameter registers, such as `DMAx_CONFIG` and `DMAx_X_COUNT`.
- control/register registers, `DMAx_IRQ_STATUS` and `DMAx_PERIPHERAL_MAP`.

The letter x in DMAx represents a specific DMA capable peripheral. For example, for DMA with default channel mapping, `DMA6_CONFIG` represents the `DMA_CONFIG` register for the UART0 receive peripheral. For default DMA channel mappings, see Table 9-5 on page 9-24.

Only parameter registers can be loaded directly from descriptor elements; descriptor elements are listed in Table 9-3 on page 9-7.

Table 9-2 on page 9-4 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, the register category, and reset value.

Table 9-2. Generic Name of DMA Memory-Mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category	Reset Value
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter	Undefined
0x04	START_ADDR	Start address of current buffer	Parameter	Undefined
0x08	DMA_CONFIG	DMA Configuration register, including enable bit	Parameter	0x0000
0x0C	Reserved	Reserved		
0x10	X_COUNT	Inner loop count	Parameter	Undefined
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter	Undefined
0x18	Y_COUNT	Outer loop count (2D only)	Parameter	Undefined

Table 9-2. Generic Name of DMA Memory-Mapped Registers (Cont'd)

MMR Offset	Generic MMR Name	MMR Description	Register Category	Reset Value
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter	Undefined
0x20	CURR_DESC_PTR	Current Descriptor Pointer	Current	Undefined
0x24	CURR_ADDR	Current DMA Address	Current	Undefined
0x28	IRQ_STATUS	interrupt status register Contains Completion and DMA Error interrupt status and channel state (Run/Fetch/Paused)	control/ register	0x0000
0x2C	PERIPHERAL_MAP	Peripheral to DMA Channel Mapping Contains a 4-bit value specifying the peripheral to associate with this DMA channel (Read-only for MDMA channels)	control/ register	See Table 9-5 on page 9-24.
0x30	CURR_X_COUNT	Current count (1D) or intra-row X count (2D), counts down from X_COUNT	Current	Undefined
0x34	Reserved	Reserved		
0x38	CURR_Y_COUNT	Current row count (2D only), counts down from Y_COUNT	Current	Undefined
0x3C	Reserved	Reserved		

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers: `NEXT_DESC_PTR`, `START_ADDR`, `CURR_DESC_PTR`, `CURR_ADDR`.

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Naming Conventions for DMA MMRs

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses the naming conventions in Table 9-3 on page 9-7, where:

The left column lists the generic name of the MMR, which is used when discussing the general operation of the DMA engine.

Note the generic names in the left column are not actually mapped to resources in the processor.

The middle column lists the specific MMR name. Only specific MMR names are mapped to processor resources.

In DMAx, the letter x represents the number of the DMA channel. For instance, `DMA3_IRQ_STATUS` is the `IRQ_STATUS` MMR for DMA Channel #3.

The channel number can be assigned by default or can be programmed. For the DMA channel numbers and the default peripheral mapping, see Table 9-5 on page 9-24.

The last column lists the macro assigned to each descriptor element in memory.

The macro name in the last column serves only to clarify the discussion of how the DMA engine operates.

Table 9-3. Naming Conventions: DMA MMRs and Descriptor Elements

Generic MMR Name	Specific MMR Name (x = DMA Channel Number)	Name of Corresponding Descriptor Element in Memory
DMA_CONFIG	DMAx_CONFIG	DMACFG
NEXT_DESC_PTR	DMAx_NEXT_DESC_PTR	NDPH (upper 16 bits), NDPL (lower 16 bits)
START_ADDR	DMAx_START_ADDR	SAH (upper 16 bits), SAL (lower 16 bits)
X_COUNT	DMAx_X_COUNT	XCNT
Y_COUNT	DMAx_Y_COUNT	YCNT
X_MODIFY	DMAx_X_MODIFY	XMOD
Y_MODIFY	DMAx_Y_MODIFY	YMOD
CURR_DESC_PTR	DMAx_CURR_DESC_PTR	N/A
CURR_ADDR	DMAx_CURR_ADDR	N/A
CURR_X_COUNT	DMAx_CURR_X_COUNT	N/A
CURR_Y_COUNT	DMAx_CURR_Y_COUNT	N/A
IRQ_STATUS	DMAx_IRQ_STATUS	N/A
PERIPHERAL_MAP	DMAx_PERIPHERAL_MAP	N/A

Naming Conventions for Memory DMA Registers

The names of memory DMA registers differ somewhat from the names of other DMA registers. memory DMA streams cannot be reassigned to different channels, whereas the peripherals associated with DMA can be mapped to any DMA channel between 0 and 19.

The processors have two DMA controller channels. Each DMA controller contains two memory DMA streams. The letter “x” denotes which of the DMA controllers the channel is in, and letters “yy” have four possible values:

- S0, memory DMA Source Stream 0
- D0, memory DMA Destination Stream 0
- S1, memory DMA Source Stream 1
- D1, memory DMA Destination Stream 1

Table 9-4 on page 9-9 shows the naming conventions for memory DMA registers.

Table 9-4. Naming Conventions for Memory DMA Registers

Generic MMR Name	Memory DMA MMR Name (yy = S0, S1, D0, or D1 x = 0 or 1 denoting DMA Controller 0 or 1)	Name of Corresponding Descriptor Element in Memory
DMA_CONFIG	MDMAx_yy_CONFIG	DMACFG
NEXT_DESC_PTR	MDMAx_yy_NEXT_DESC_PTR	NDPH (upper 16 bits), NDPL (lower 16 bits)
START_ADDR	MDMAx_yy_START_ADDR	SAH (upper 16 bits), SAL (lower 16 bits)
X_COUNT	MDMAx_yy_X_COUNT	XCNT
Y_COUNT	MDMAx_yy_Y_COUNT	YCNT
X_MODIFY	MDMAx_yy_X_MODIFY	XMOD
Y_MODIFY	MDMAx_yy_Y_MODIFY	YMOD
CURR_DESC_PTR	MDMAx_yy_CURR_DESC_PTR	N/A
CURR_ADDR	MDMAx_yy_CURR_ADDR	N/A
CURR_X_COUNT	MDMAx_yy_CURR_X_COUNT	N/A
CURR_Y_COUNT	MDMAx_yy_CURR_Y_COUNT	N/A
IRQ_STATUS	MDMAx_yy_IRQ_STATUS	N/A
PERIPHERAL_MAP	MDMAx_yy_PERIPHERAL_MAP	N/A

Next Descriptor Pointer (DMAx_NEXT_DESC_PTR / MDMAx_yy_NEXT_DESC_PTR) Registers

The NEXT_DESC_PTR register specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used only in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, the 32-bit NEXT_DESC_PTR is copied into CURR_DESC_PTR. Then, during the descriptor fetch, the CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, NEXT_DESC_PTR, and not CURR_DESC_PTR, must be programmed directly via MMR access before starting DMA operation.

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

Next Descriptor Pointer Register (DMAx_NEXT_DESC_PTR / MDMAx_yy_NEXT_DESC_PTR)
R/W prior to enabling channel, RO after enabling channel

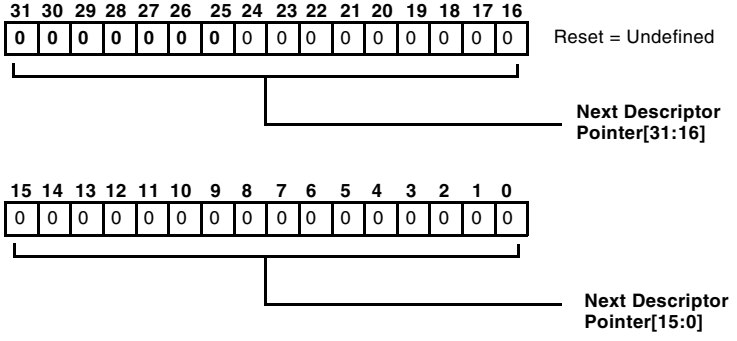


Figure 9-1. Next Descriptor Pointer Register

Start Address Register (DMAx_START_ADDR/MDMAx_yy_START_ADDR)

The START_ADDR register, shown in Figure 9-2 on page 9-11, contains the start address of the data buffer currently targeted for DMA.

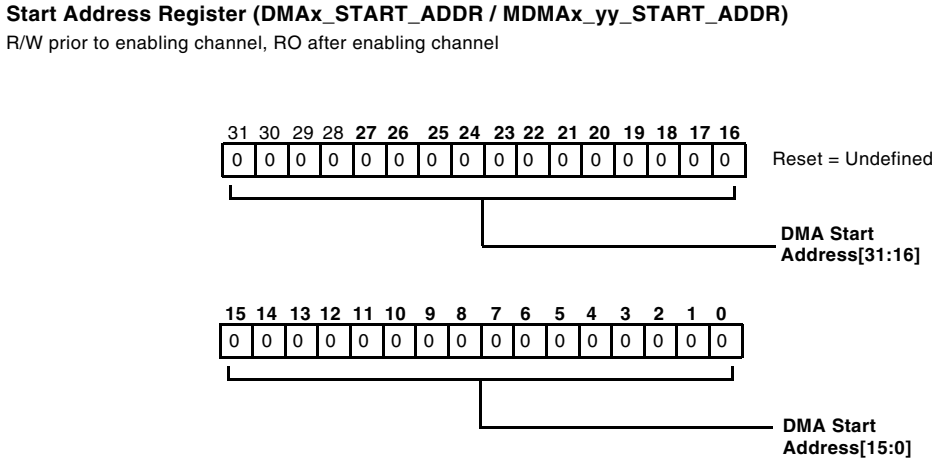


Figure 9-2. Start Address Register

DMA Configuration Register (DMAx_CONFIG / MDMAx_yy_CONFIG)

The DMA_CONFIG register, shown in Figure 9-3 on page 9-12, is used to set up DMA parameters and operating modes. Note that writing the DMA_CONFIG register while DMA is already running will cause a DMA error unless writing with the DMA_EN bit set to 0.

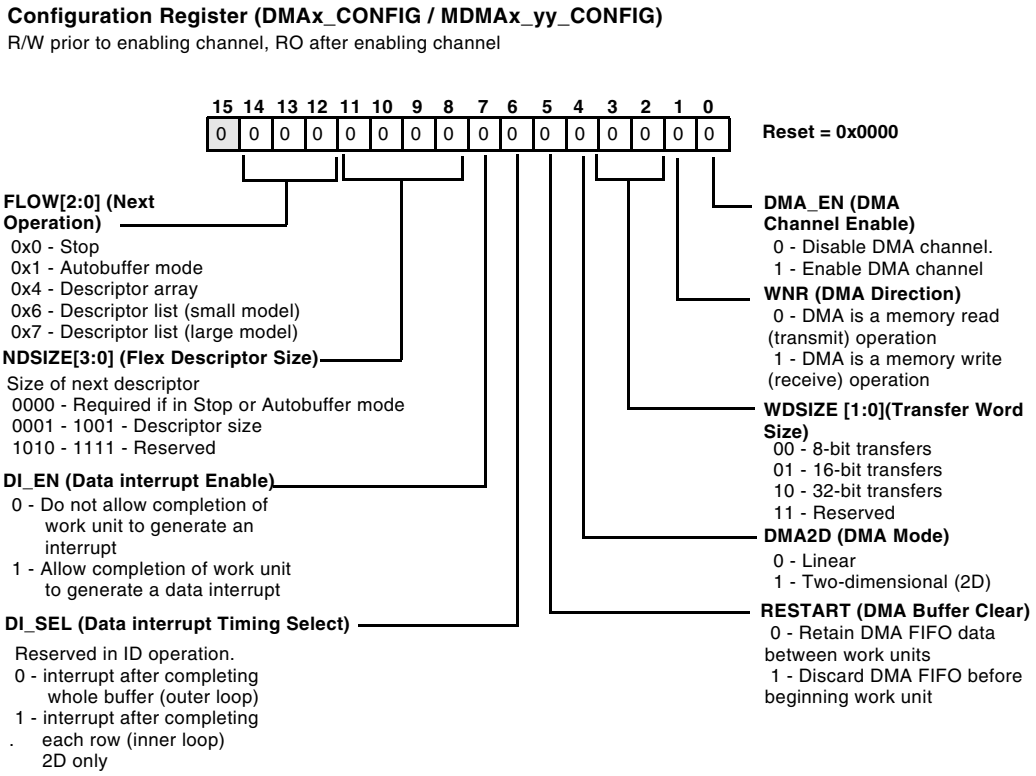


Figure 9-3. Configuration Register

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

FLOW[2:0] (Next Operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:

0x0 - Stop. When the current work unit completes, the DMA channel stops automatically, after signalling an interrupt (if selected). The DMA_RUN status bit in DMAx_IRQ_STATUS changes from 1 to 0, while the DMA_EN bit in DMAx_CONFIG is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to DMAx_CONFIG specifying the next work unit, in which DMA_EN is set to 1.

0x1 - Autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular-buffer fashion based on user-programmed DMAx MMR settings. On completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMA_EN bit in DMAx_CONFIG.

0x4 - Descriptor Array mode. This mode fetches a descriptor from memory that does not include the Next Descriptor Pointer High (NDPH) or Next Descriptor Pointer Low (NDPL) elements. Because the descriptor does not contain a next descriptor pointer entry, the DMA engine defaults to using CURR_DESC_PTR to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.

0x6 - Descriptor list (small model) mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the next descriptor pointer field are taken from the upper 16 bits of the NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

0x7 - Descriptor list (large model) mode. This mode fetches a descriptor from memory that includes NDPH and NDPL, thus allowing maximum flexibility in locating descriptors in memory.

NDSIZE[3:0] (flex descriptor size). This field specifies the number of DMA MMRs to load from descriptor elements in memory. This field must be 0 if in Stop or Autobuffer mode. If NDSIZE and FLOW specify a descriptor that extends beyond Y-Modify (YMOD), a DMA Error results.

DI_EN (data interrupt enable). This bit specifies whether to allow completion of a work unit to generate a data interrupt.

DI_SEL (data interrupt timing select). This bit specifies the timing of a data interrupt: after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2D DMA operation.

RESTART (DMA buffer clear). This bit specifies whether receive data held in the channel's data FIFO should be preserved (RESTART=0) or discarded (RESTART=1) before beginning the next work unit. Receive data is automatically discarded when DMA_EN changes from 0 to 1, typically when a channel is first enabled. Received FIFO data should usually be retained between work units if the work units make up a continuous data stream. If, however, a new work unit starts a new data stream, the RESTART bit should be set to 1 to clear out any previously received data.

DMA2D (DMA mode). This bit specifies whether DMA mode involves only X_COUNT and X_MODIFY (one-dimensional DMA) or also involves Y_COUNT and Y_MODIFY (two-dimensional DMA).

WDSIZE[1:0] (Transfer Word Size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port), or through the 16-bit DAB bus. The DMA address pointer increment sizes (strides) must be a multiple of the transfer unit size: 1 for 8-bit, 2 for 16-bit, 4 for 32-bit.

For information about how to set up each DMA channel to support different transfer widths, see "Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers" on page 9-22.

WNR (DMA Direction). This bit specifies DMA direction: memory read (0) or memory write (1).

DMA_EN (DMA Channel Enable). This bit specifies whether to enable a given DMA channel.

When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

Inner Loop Count (DMAx_X_COUNT, MDMAx_yy_X_COUNT) Registers

For 2D DMA, the X_COUNT registers, shown in Figure 9-4 on page 9-15, contain the inner loop count. For 1D DMA, it specifies the number of elements to read in. For details, see "Two-Dimensional DMA" on page 9-33. A value of 0 in X_COUNT corresponds to 65,536 elements.

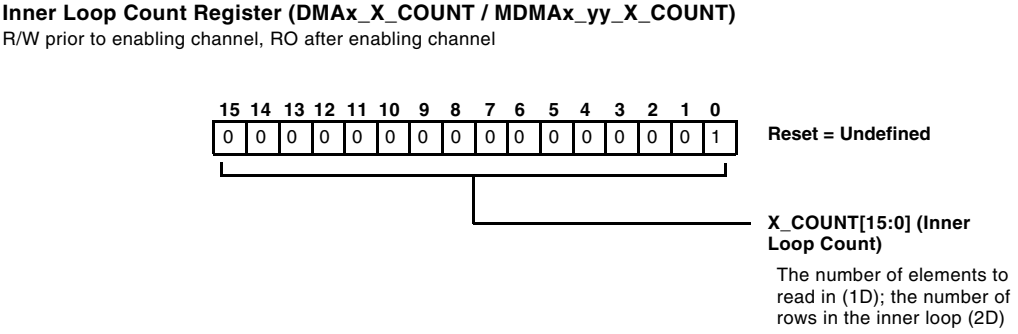


Figure 9-4. Inner Loop Count Register

Inner Loop Address Increment (DMAx_X_MODIFY, MDMAx_yy_X_MODIFY) Registers

The inner loop address increment (X_MODIFY) registers contain a signed, 2's complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

In 2D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, Y_MODIFY is applied instead, except on the very last transfer of each work unit. X_MODIFY is always applied on the last transfer of a work unit.

The X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

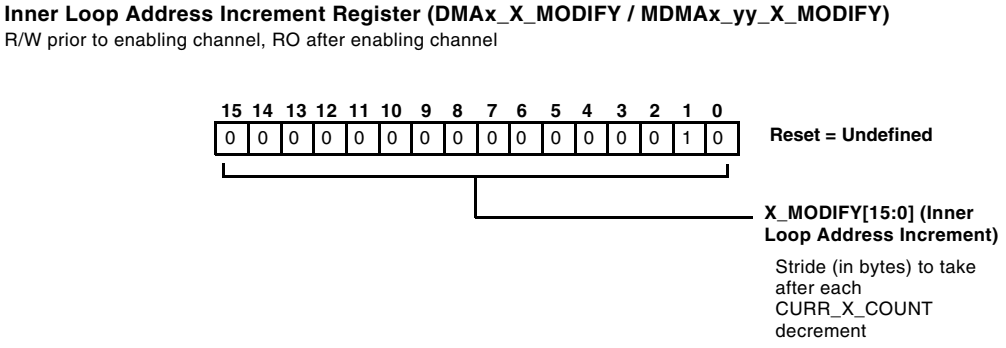


Figure 9-5. Inner Loop Address Increment Register

Outer Loop Count (DMAx_Y_COUNT, MDMAx_yy_Y_COUNT) Registers

For 2D DMA, the outer loop count (`Y_COUNT`) registers contain the outer loop count. It is not used in 1D DMA mode. This register contains the number of rows in the outer loop of a 2D DMA sequence. For details, see "Two-Dimensional DMA" on page 9-33.

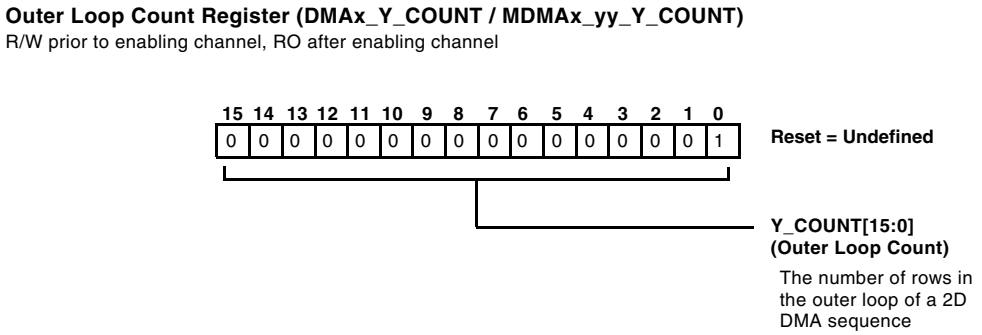


Figure 9-6. Outer Loop Count Register

Outer Loop Address Increment (DMAx_Y_MODIFY, MDMAx_yy_Y_MODIFY) Registers

The outer loop address increment (`Y_MODIFY`) registers contain a signed, 2's complement value. This byte-address increment is applied after each decrement of `CURR_Y_COUNT` except for the last item in the 2D array on which the `CURR_Y_COUNT` also expires. The value is the offset between the last word of one "row" and the first word of the next "row." For details, see "Two-Dimensional DMA" on page 9-33.

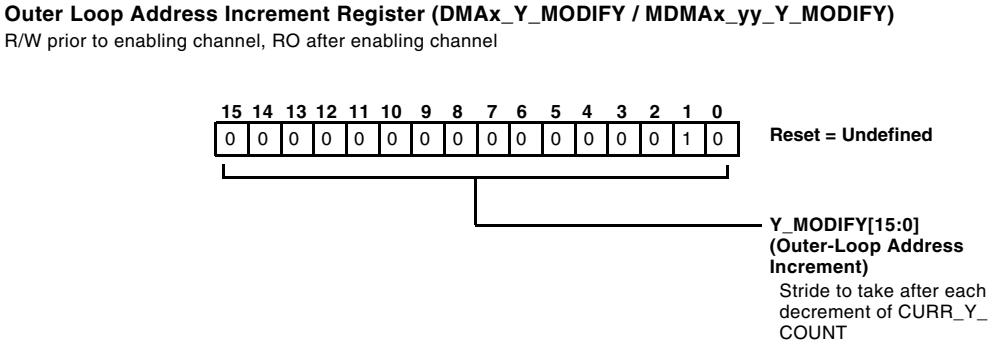


Figure 9-7. Outer Loop Address Increment Register

Current Descriptor Pointer (DMAx_CURR_DESC_PTR, MDMAx_yy_CURR_DESC_PTR) Registers

The current descriptor pointer (CURR_DESC_PTR) registers contain the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW=4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (FLOW=6 or 7), this register is initialized from NEXT_DESC_PTR before loading each descriptor. Then, the address in CURR_DESC_PTR increments as each descriptor element is read in. When the entire descriptor has been read, CURR_DESC_PTR contains this value:

```
Descriptor Start Address + Descriptor Size (# of elements)
```

For descriptor array mode (FLOW=4), this register, and not the NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

Do not use the CURR_DESC_PTR registers for software synchronization with an ongoing DMA operation. See the "Synchronization of Software and DMA" on page 9-43 for details.

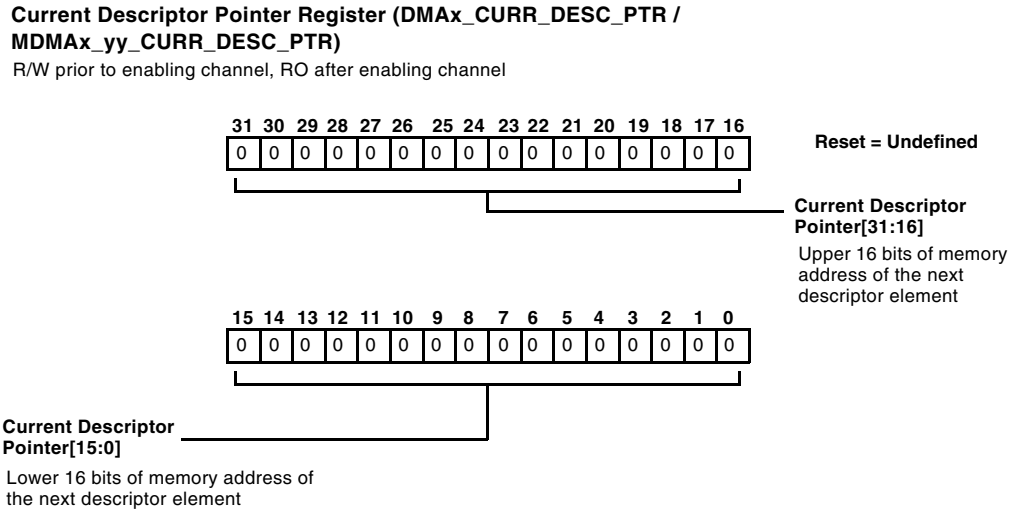


Figure 9-8. Current Descriptor Pointer Register

Current Address (DMAx_CURR_ADDR, MDMAx_yy_CURR_ADDR) Registers

The current address (CURR_ADDR) registers, shown in Figure 9-9 on page 9-20, contain the present DMA transfer address for a given DMA session. At the start of a DMA session, it is loaded from the START_ADDR register, and it is incremented as each transfer occurs. The Current Address register contains 32 bits.

Do not use the CURR_ADDR registers for software synchronization with an ongoing DMA operation. See the "Synchronization of Software and DMA" on page 9-43 for details.

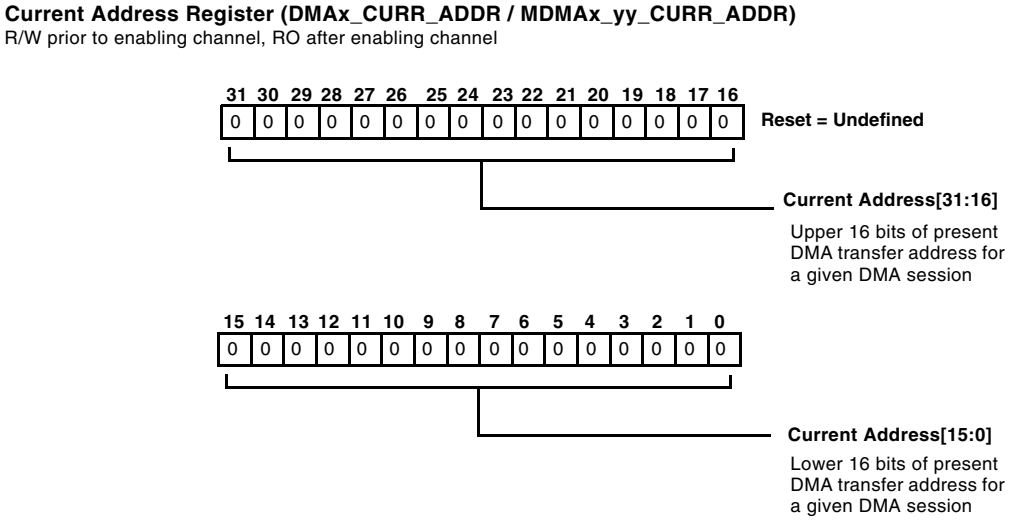


Figure 9-9. Current Address Register

Current Inner Loop Count (DMAx_CURR_X_COUNT, MDMAx_yy_CURR_X_COUNT) Registers

The current inner loop count (CURR_X_COUNT) register is loaded by X_COUNT at the beginning of each DMA session (for 1D DMA) and also after the end of DMA for each row (for 2D DMA). Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2D DMA, CURR_X_COUNT is 0 only when the entire transfer is complete. Between rows it is equal to X_COUNT.

Do not use the CURR_X_COUNT registers for software synchronization with an ongoing DMA operation. See the "Synchronization of Software and DMA" on page 9-43 for details.

Current Inner Loop Count Register (DMAx_CURR_X_COUNT /

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

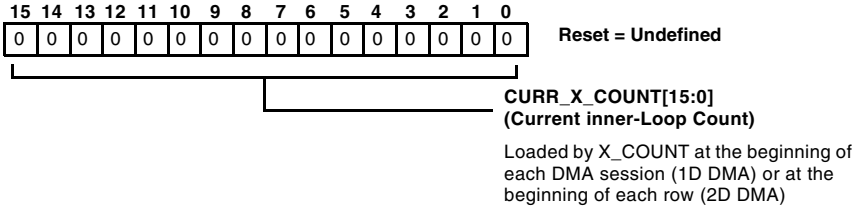


Figure 9-10. Current Inner Loop Count Register

Current Outer Loop Count Register (DMAx_CURR_Y_COUNT / MDMAx_yy_CURR_Y_COUNT)

The current outer loop count (CURR_Y_COUNT) register is loaded by Y_COUNT at the beginning of each 2D DMA session. It is not used for 1D DMA. This register is decremented each time that the CURR_X_COUNT register expires during 2D DMA operation (1 to X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2D DMA session is complete, CURR_Y_COUNT=1 and CURR_X_COUNT=0.

Do not use the CURR_Y_COUNT registers for software synchronization with an ongoing DMA operation. See the "Synchronization of Software and DMA" on page 9-43 for details.

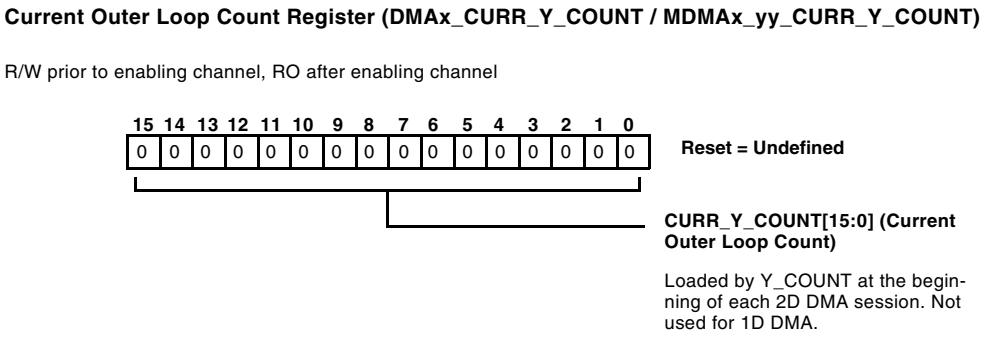


Figure 9-11. Current Outer Loop Count Register

Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers

Each DMA channel PERIPHERAL_MAP register contains bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel.

There are two sets of PERIPHERAL_MAP registers. One set is for the channels associated with DMA controller 0 and the other set for the channels associated with DMA controller 1. Peripherals are assigned to a specific controller (0 or 1). This assignment is fixed and is not selectable by the user.

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

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

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7.
2. Write `DMA6_PERIPHERAL_MAP` with `0x7000` and `DMA7_PERIPHERAL_MAP` with `0x6000`.
3. Enable DMA on channels 6 and/or 7.

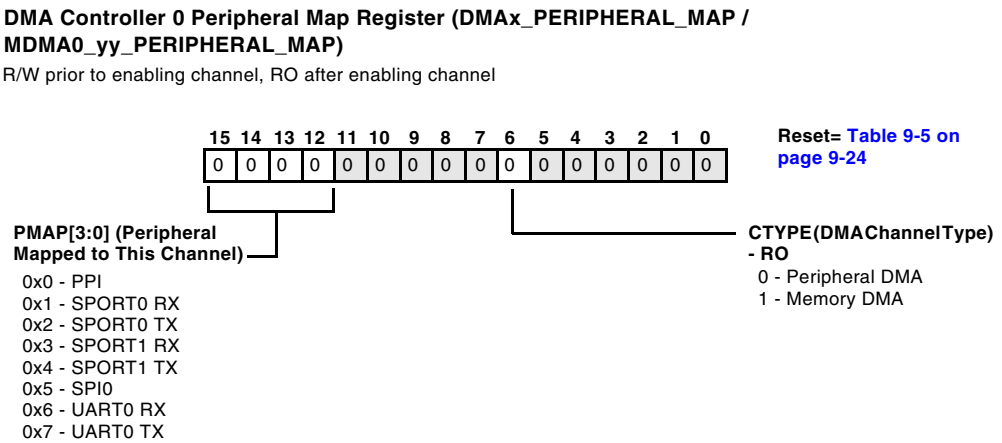


Figure 9-12. DMA Controller 0 Peripheral Map Register

Table 9-5 on page 9-24 lists the binary peripheral map settings for each DMA capable peripheral.

Table 9-5. Peripheral Mapping of DMA Controller 0

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting	Comments
DMA0 (highest priority)	PPI	b#0000 0000 0000 0000	
DMA1	SPORT0 RX	b#0001 0000 0000 0000	
DMA2	SPORT0 TX	b#0010 0000 0000 0000	
DMA3	SPORT1 RX	b#0011 0000 0000 0000	
DMA4	SPORT1 TX	b#0100 0000 0000 0000	
DMA5	SPI0	b#0101 0000 0000 0000	
DMA6	UART0 RX	b#0110 0000 0000 0000	
DMA7	UART0 TX	b#0111 0000 0000 0000	
MDMA0_D0	Mem DMA Stream 0 TX (destination)	b#0000 0000 0100 0000	Not re-assignable
MDMA0_S0	Mem DMA Stream 0 RX (source)	b#0000 0000 0100 0000	Not re-assignable
MDMA0_D1	Mem DMA Stream 1 TX (destination)	b#0000 0000 0100 0000	Not re-assignable
MDMA0_S1 (lowest priority)	Mem DMA Stream 1 RX (source)	b#0000 0000 0100 0000	Not re-assignable

The MemDMA streams in DMA Controller 0, though the lowest priority for DMA controller 0, have priority over the peripheral DMA channels on DMA Controller 1 because DMA Controller 0 has higher priority than DMA Controller 1.

Table 9-6 on page 9-25 assumes the following peripheral ID mappings into the 4-bit PMAP field:

Table 9-6. Peripheral ID Mappings Into PMAP Field for DMA Controller 0

Peripheral	PMAP ID Number
PPI	b#0000
SPORT0 RX	b#0001
SPORT0 TX	b#0010
SPORT1 RX	b#0011
SPORT1 TX	b#0100
SPI	b#0101
UART0 RX	b#0110
UART0 TX	b#0111

DMA Controller 1 Peripheral Map Register (DMAx_PERIPHERAL_MAP / MDMAx_yy_PERIPHERAL_MAP)

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

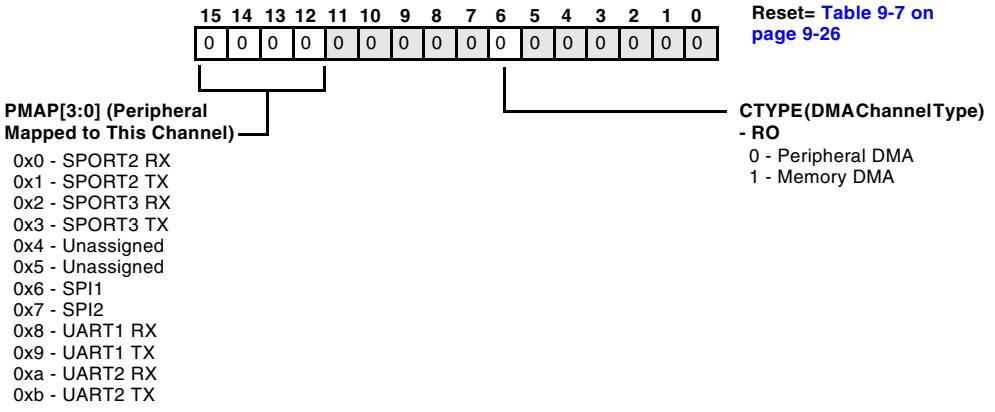


Figure 9-13. DMA Controller 1Peripheral Map Register

Table 9-7. Peripheral Mapping of DMA Controller 1

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting	Comments
DMA8 (highest priority)	SPORT2 RX	b#0000 0000 0000 0000	
DMA9	SPORT2 TX	b#0001 0000 0000 0000	
DMA10	SPORT3 RX	b#0010 0000 0000 0000	
DMA11	SPORT3 TX	b#0011 0000 0000 0000	
DMA12	Unassigned	b#0100 0000 0000 0000	
DMA13	Unassigned	b#0101 0000 0000 0000	
DMA14	SPI1	b#0110 0000 0000 0000	
DMA15	SPI2	b#0111 0000 0000 0000	
DMA16	UART1 RX	b#1000 0000 0000 0000	
DMA17	UART1 TX	b#1001 0000 0000 0000	

Table 9-7. Peripheral Mapping of DMA Controller 1

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting	Comments
DMA18	UART2 RX	b#1010 0000 0000 0000	
DMA19	UART2 TX	b#1011 0000 0000 0000	
MDMA1_D0	Mem DMA Stream 2 TX (destination)	b#0000 0000 0100 0000	Not re-assign-able
MDMA1_S0	Mem DMA Stream 2 RX (source)	b#0000 0000 0100 0000	Not re-assign-able
MDMA1_D1	Mem DMA Stream 3 TX (destination)	b#0000 0000 0100 0000	Not re-assign-able
MDMA1_S1 (lowest priority)	Mem DMA Stream 3 RX (source)	b#0000 0000 0100 0000	Not re-assign-able

Table 9-8. Peripheral ID Mappings Into PMAP Field for DMA Controller 1

Peripheral	PMAP ID Number
SPORT2 RX	b#0000
SPORT2 TX	b#0001
SPORT3 RX	b#0010
SPORT3 TX	b#0011
Unassigned	b#0100
Unassigned	b#0101
SPI1	b#0110
SPI2	b#0111
UART1 RX	b#1000
UART1 TX	b#1001

Table 9-8. Peripheral ID Mappings Into PMAP Field for DMA Controller 1

Peripheral	PMAP ID Number
UART2 RX	b#1010
UART2 TX	b#1011

Interrupt Status Register (DMAx_IRQ_STATUS / MDMAx_yy_IRQ_STATUS)

The interrupt register (`IRQ_STATUS`) register, shown in Figure 9-14 on page 9-29, contains bits that record whether the DMA channel:

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

Note the `DMA_DONE` interrupt is asserted when the last memory access (read or write) has completed. For a transmit (memory read) transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the `DMA_RUN` bit. As long as there is undelivered transmit data in the FIFO, `DMA_RUN` is 1.

Interrupt Status Register (DMAx_IRQ_STATUS / MDMAx_yy_IRQ_STATUS)

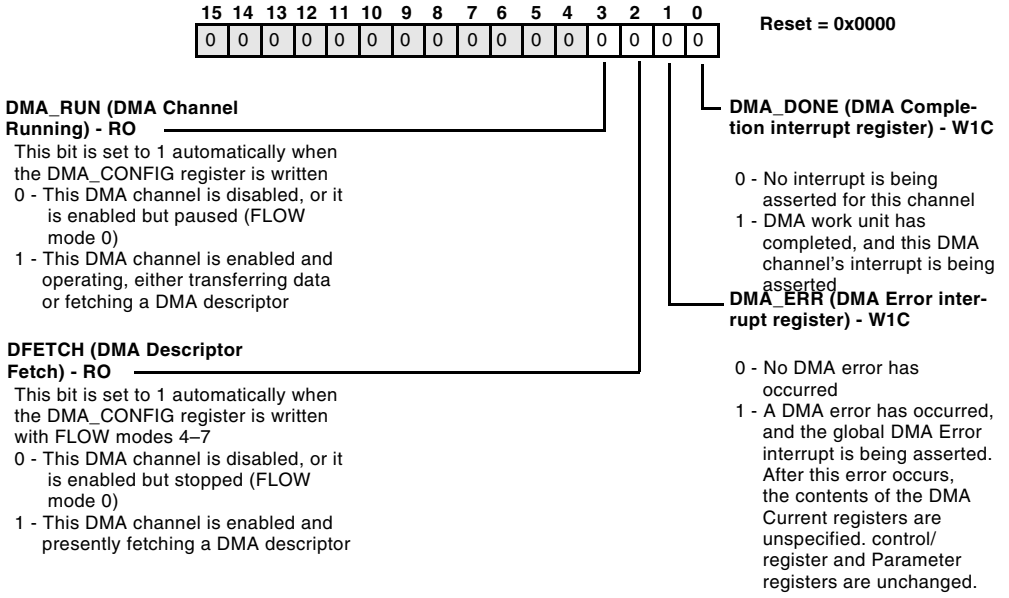


Figure 9-14. Interrupt Status Register

The processor supports a flexible interrupt control structure with three interrupt sources. Separate IRQ levels are allocated for data, peripheral errors, and DMA errors.

- data-driven interrupts (see Table 9-7 on page 9-26).
- peripheral error interrupts.
- DMA error interrupts (for example, bad descriptor or bus error)

Table 9-9. Data Driven Interrupts

Interrupt Name	Description
No interrupt	Interrupts can be disabled for a given work unit.
Peripheral interrupt	Peripheral (non-DMA) interrupt.
Row Completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer Completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

All DMA channels are OR'ed together into one system-level DMA Error interrupt on each DMA controller. The individual IRQ_STATUS words of each channel can be read to identify the channel that caused the DMA Error interrupt.

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

Flex Descriptor Structure

DMA flex descriptors are variable sized data structures whose contents are loaded into DMA Parameter registers. The sequence of registers in the descriptor is essentially fixed (among three similar variations), but the length of the descriptor is completely programmable. The DMA channel registers are ordered so that the registers that are most commonly reloaded per work unit are at the lowest MMR addresses. The user may choose whether or not to use descriptors. If not using descriptors, the user can write the DMA MMRs directly to start DMA, and use either Autobuffer mode for continuous operation or Stop mode for single-buffer operation.

To use descriptors, the user programs the NDSIZE field of the DMAx_CONFIG register with the number of DMA registers to load from the descriptor, starting with the lowest MMR address. The user may select a descriptor size from one entry (the lower 16 bits of START_ADDR) to nine entries (all the DMA parameters).

The three variations of the descriptor value sequences depend on whether a next descriptor pointer is included and, if so, what kind.

1. None included (descriptor array mode)
2. The lower 16 bits of the next descriptor pointer (descriptor list, small model)
3. All 32 bits of the next descriptor pointer (descriptor list, large model)

All the other registers not loaded from the descriptor retain their prior values (although the CURR_ADDR, CURR_X_COUNT, CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.)

There are certain DMA settings that are not allowed to change from one descriptor to the next in a chain (small or large list and array modes). These include DMA direction, word size, and memory space (that is, switching between internal and external memory).

A single descriptor chain cannot control the transfer of a sequence of data buffers which reside in different memory spaces. Instead, group the data buffers into chains of buffers in the same space, but do not link the chains together. Transfer the first chain, wait for its final interrupt, and then start the next chain with an MMR write to `DMA_CONFIG`.

Note that while the user must locate each chain's data buffers in the same memory space, the descriptor structures themselves may be placed in any memory space, and they may link from a descriptor in one space to a descriptor in the other space without restriction.

Table 9-8 on page 9-27 shows the descriptor offsets for descriptor elements in the three modes described above. Note the names in the table list the descriptor elements in memory, not the actual MMRs into which they are eventually loaded.

Figure 9-15. Descriptor Element Definitions

Descriptor Element	Name	Definition
NDPL	Next Descriptor Pointer Low	Lower 16 Bits of Next Descriptor's Address
NDPH	Next Descriptor Pointer High	Upper 16 Bits of Next Descriptor's Address
SAL	Start Address Low	Lower 16 Bits of Buffer's Start Address
SAH	Start Address High	Upper 16 Bits of Buffer's Start Address
DMACFG	DMA Configuration Word	Value written to DMAx_CONFIG
XCNT	X Count	Number of elements to transfer
XMOD	X Modify	Number of bytes in stride
YCNT	Y Count	Number of rows to transfer (2-D)
YMOD	Y Modify	Number of rows in stride (2-D)

Table 9-10. Parameter Registers and Descriptor Offsets

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

Two-Dimensional DMA

Two-dimensional DMA supports arbitrary row and column sizes up to 64 K x 64 K elements, as well as arbitrary X_MODIFY and Y_MODIFY values up to ±32 K bytes. Furthermore, Y_MODIFY can be negative, allowing implementation of interleaved data streams. X_COUNT and Y_COUNT specify the row and column sizes, where X_COUNT must be 2 or greater.

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

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

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

After the last transfer completes, CURR_Y_COUNT=1, CURR_X_COUNT=0, and CURR_ADDR is equal to the last item's address plus X_MODIFY. Note that if the DMA channel is programmed to refresh automatically (autobuffer mode), then these registers are loaded from X_COUNT, Y_COUNT, and START_ADDR upon the first data transfer.

Example 1: Retrieve a 16x8 Block of Bytes From a Video Frame Buffer of Size (N x M) Pixels:

```
X_MODIFY = 1

X_COUNT = 16

Y_MODIFY = N-15 (offset from the end of one row to the start of
another)

Y_COUNT = 8
```

This produces the following address offsets from the start address:

```
0,1,2,...15,

N,N+1, ... N+15,

2N, 2N+1,... 2N+15, ...

7N, 7N+1,... 7N+15,

0, ...
```

Example 2: Receive a Video Data Stream of Bytes, (R,G,B Pixels) x (N x M Image Size):

```
X_MODIFY = (N*M)
```

```
X_COUNT = 3

Y_MODIFY = 1 - 2(N*M) (negative)

Y_COUNT = (N*M)
```

This produces the following address offsets from the start address:

```
0, (N*M), 2(N*M),

1, (N*M)+1, 2(N*M)+1,

2, (N*M)+2, 2(N*M)+2,

    ...

(N*M)-1, 2(N*M)-1, 3(N*M)-1,

0, ...
```

DMA Operation Flow

Figure 9-16 on page 9-36 and Figure 9-17 on page 9-37 describe the DMA Flow.

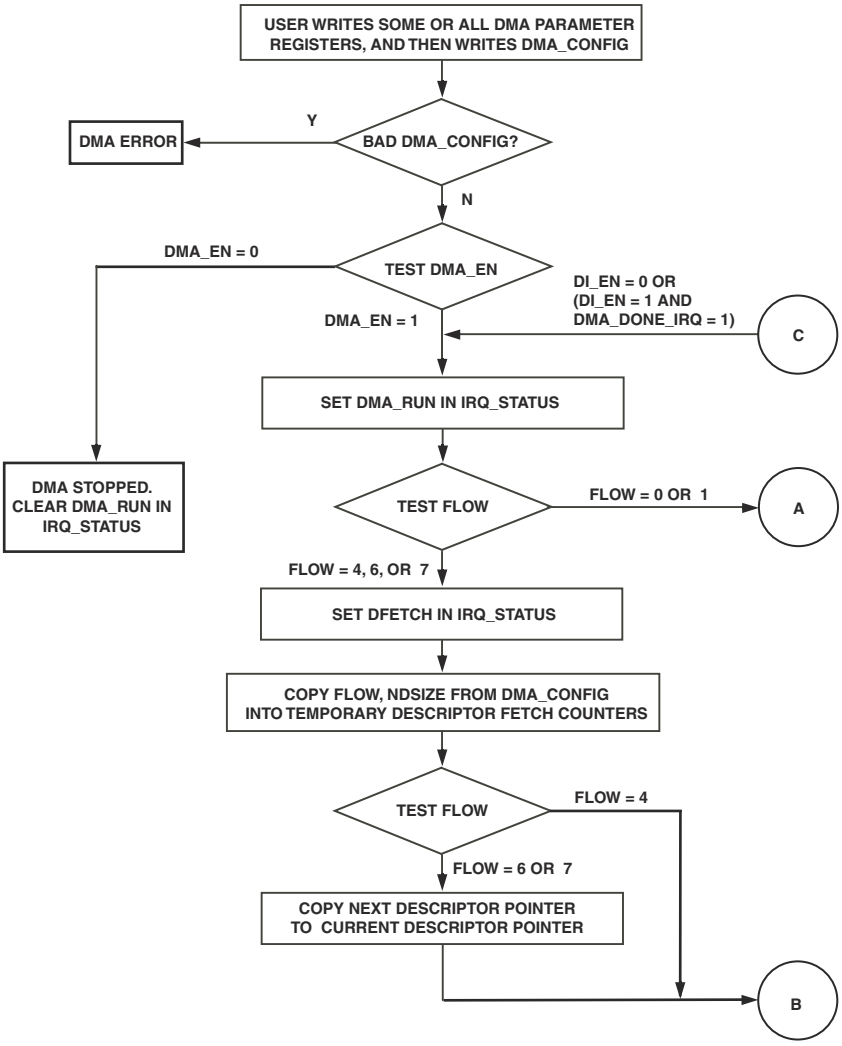


Figure 9-16. DMA Flow, from DMA Controller point of view (1 of 2)

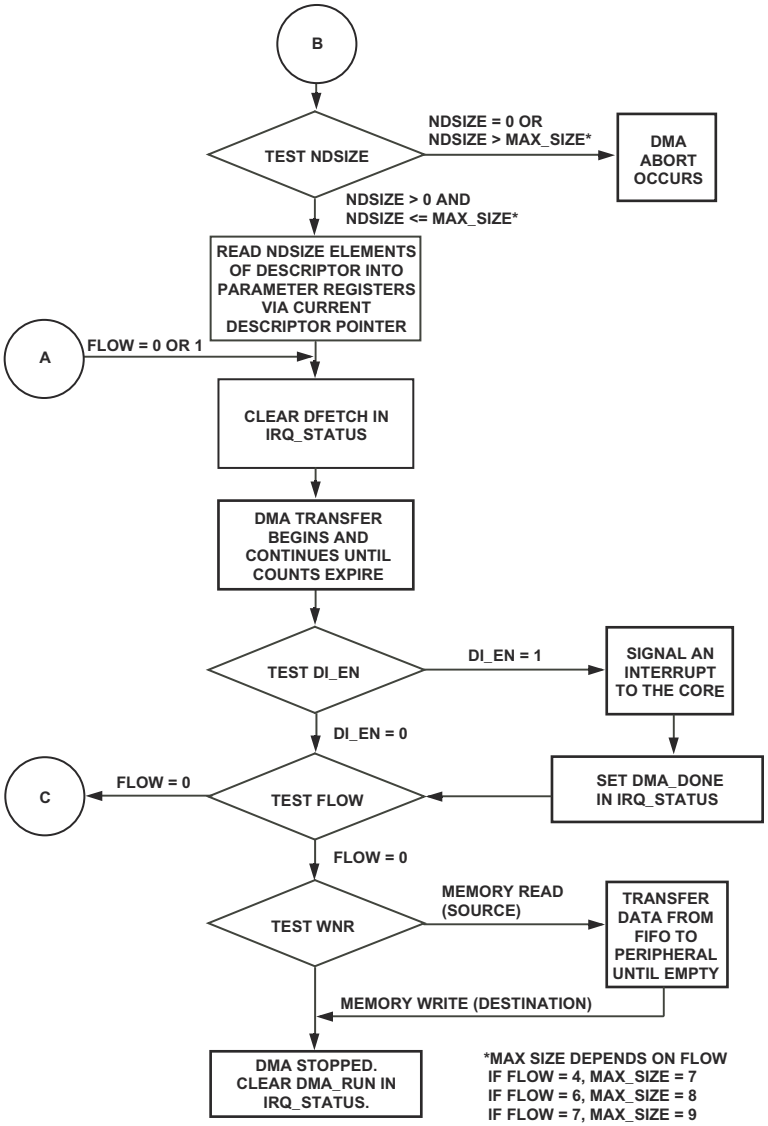


Figure 9-17. DMA Flow, from DMA Controller point of view (2 of 2)

DMA Startup

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

Before initiating DMA for the first time on a given channel, be sure to initialize all Parameter registers. Be especially careful to initialize the upper 16 bits of NEXT_DESC_PTR and START_ADDR, because they might not otherwise be accessed, depending on the chosen FLOW mode of operation.

To start DMA operation on a given channel, some or all of the DMA Parameter registers must first be written directly. At a minimum, the NEXT_DESC_PTR register (or CURR_DESC_PTR register in FLOW = 4 mode) must be written at this stage, but the user may wish to write other DMA registers that might be static throughout the course of DMA activity (for example, X_MODIFY, Y_MODIFY). The contents of NDSIZE and FLOW in DMA_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMA_CONFIG with DMA_EN = 1.

When DMA_CONFIG is written directly, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMA_EN=1, the DMACFG descriptor element that was read into DMA_CONFIG assumes control. (Before this point, the direct write to DMA_CONFIG had control.)

As Figure 9-16 on page 9-36 and Figure 9-17 on page 9-37 show, at startup the FLOW and NDSIZE bits in DMA_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more Current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values. Moreover, the reset values of DMA MMRs are never restored except after a system reset.

If the FLOW value specifies small or large descriptor list modes, the NEXT_DESC_PTR is copied into CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into NEXT_DESC_PTR, but the fetch of the current descriptor continues using CURR_DESC_PTR. After completion of the descriptor fetch, CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

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

If DMACFG is not part of the descriptor, the previous DMA_CONFIG settings (as written by MMR access at startup) control the work unit operation. If DMACFG is part of the descriptor, then the DMA_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor DMACFG and by the parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the IRQ_STATUS register indicate the state of the DMA channel. After a write to DMA_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, the current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- START_ADDR is copied to CURR_ADDR
- X_COUNT is copied to CURR_X_COUNT
- Y_COUNT is copied to CURR_Y_COUNT

Then DMA data transfer operation begins, as shown in Figure 9-17 on page 9-37.

DMA Refresh

On completion of a work unit, the DMA controller completes the transfer of all data between memory and the DMA unit. If enabled by DI_EN, the DMA controller signals an interrupt to the core and sets the DMA_DONE bit in the channel's IRQ_STATUS register.

When the FLOW bit is cleared (= 0) the operation is stopped by clearing the DMA_RUN bit in IRQ_STATUS after any transmit data in the channel's DMA FIFO has been transferred to the peripheral.

During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in IRQ_STATUS to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

If FLOW = 4 (descriptor array): Loads a new descriptor from memory into DMA registers via the contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMA_CONFIG value prior to the beginning of the fetch.

If FLOW = 6 (descriptor list small): Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The first descriptor element loaded is a new 16-bit value for the lower 16 bits of NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR retain their former value. This supports a shorter, more efficient descriptor than the descriptor list large model, suitable whenever the application can place the channel's descriptors in the same 64KB range of memory.

If FLOW = 7 (descriptor list large): Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing

CURR_DESC_PTR. The first descriptor elements loaded are a new 32-bit value for the full NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory, external memory, or ROM.

Note if it is necessary to link from a descriptor chain whose descriptors are in one 64KB area to another chain whose descriptors are outside that area, only one descriptor needs to use FLOW=7–just the descriptor which contains the link leaving the 64KB range. All the other descriptors located together in the same 64KB areas may use FLOW = 6.

If FLOW = 1, 4, 6, or 7 (autobuffer, descriptor array, descriptor list small, or descriptor list large, respectively):

(Re)loads the current registers:

CURR_ADDR loaded from START_ADDR,

CURR_X_COUNT loaded from X_COUNT,

CURR_Y_COUNT loaded from Y_COUNT

The DFETCH bit in IRQ_STATUS is then cleared, after which the DMA transfer begins again, as shown in Figure 9-17 on page 9-37.

To Stop DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMA_EN bit is 0. A write of 0 to the entire register always terminates DMA gracefully (without DMA abort).

Before enabling the channel again, make sure that any slow memory read operations that may have started are completed (for example, reads from slow external memory). Do not enable the channel again until any such reads are complete.

Software Management of DMA

Several synchronization and control methods are available for use in development of software tasks which manage DMA and MDMA (see also "Memory DMA" on page 9-53). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each DMA peripheral and MDMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the IRQ_STATUS register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of IRQ_STATUS, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note since every interrupt channel has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Polling of the CURR_ADDR, CURR_DESC_PTR, or CURR_X/Y_COUNT registers is not recommended as a method of precisely synchronizing DMA with data processing, due to DMA FIFOS and DMA/memory pipelining. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA receive (memory write) operation to external memory, assume a DMA write by channel A is initiated which causes the SRAM to perform a page open operation which will take many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but which will be stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by the channel B CURR_ADDR has or has not been written, based on examination of the CURR_ADDR register contents.

Polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software, however, if allowances are made for the lengths of the DMA/memory pipeline. The length of the DMA FIFO is four locations (either four 8-, or 16-bit data elements, or two 32-bit data elements) for a peripheral DMA channel and eight locations (or four 32-bit data elements) for an MDMA FIFO. The DMA does not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished). Next, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements, while the length of the DMA and EBIU pipelines is approximately three data elements, measuring from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. (Note this is a maximum, as the DMA/memory pipeline may include traffic from other DMA channels.) For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its `CURR_X_COUNT` register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. The total pipeline length is no greater than the sum of 4 (for the PDMA FIFO) plus 6 (for the DMA/memory pipeline), or 10 data elements, so it is safe to conclude that the DMA transfer of the first 40-10 = 30 data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's `IRQ_STATUS` register to confirm completion of DMA, rather than polling current-address/pointer/count registers. When the DMA system issues an interrupt or changes an `IRQ_STATUS` status bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to DSP code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO; for memory write DMA, the DMA unit will have received an acknowledge from L1 or the EBIU that the data has been written.

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

Single-buffer DMA Transfers

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

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

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering (FLOW=1) may be an effective option. Here, DMA is transferred from or to a memory buffer using a circular addressing scheme, using either 1- or 2-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

1-D, interrupt driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.

2-D, interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set DI_SEL = 1 in DMA_CONFIG) to be signaled at the completion of each DMA inner loop. For example, two 512-word sub-buffers inside a 1 K-word buffer could be used to receive 16-bit peripheral data with these settings:

START_ADDR = buffer base address

DMA_CONFIG = 0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1, DMA2D = 1, WDSIZ = 01, WNR = 1, DMA_EN = 1)

X_COUNT = 512

X_MODIFY = 2 for 16-bit data

Y_COUNT = 2 for two sub-buffers

Y_MODIFY = 2, same as X_MODIFY for contiguous sub-buffers

In this way, a traditional double-buffer or “ping-pong” scheme could be implemented.

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

START_ADDR = buffer base address

DMA_CONFIG = 0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1, WDSIZ = 10, WNR = 1, DMA_EN = 1)

X_COUNT = 16

X_MODIFY = 4 for 32-bit data

`Y_COUNT = 4` for four sub-buffers

`Y_MODIFY = 4`, same as `X_MODIFY` for contiguous sub-buffers

The synchronization core might read `Y_COUNT` to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of `Y_COUNT` shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.

1-D unsynchronized FIFO—If the system design guarantees that the processing of peripheral data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1-D autobuffer-mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1-D or 2-D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory-pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list as desired by selecting the appropriate `FLOW` setting in `DMA_CONFIG`.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

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

The lower byte of DMA_CONFIG specifies the DMA transfer to be performed by the current descriptor (for example, interrupt-enable, 2D-mode)

The upper byte of DMA_CONFIG specifies the format of the next descriptor in the chain. The NDSIZ and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the DMA_CONFIG value written to the DMA channel's DMA_CONFIG register should correspond to the current descriptor. At a minimum, the FLOW, NDSIZ, WNR, and DMA_EN fields must all agree with the current descriptor; the WDSIZ, DI_EN, DI_SEL, RESTART, and DMA2D fields will be taken from the DMA_CONFIG value in the descriptor read from memory (and the field values initially written to the register are ignored).

Descriptor Queue Management

A system designer might want to write a DMA Manager facility which accepts DMA requests from other software. The DMA Manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's NDPTR points to the next, and the last descriptor points to the first.

The code which writes into this descriptor list could use the processor's circular addressing modes (Ix, Lx, Mx, and Bx registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the NDPTR members of each descriptor could even be written once at startup, and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

1. interrupt on every descriptor
2. interrupt minimally - only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

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

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a `DMA_CONFIG` value with a `FLOW` value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts equal), the software increments its count and then starts the DMA unit by writing the new descriptor's `DMA_CONFIG` value to the DMA channel's `DMA_CONFIG` register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's DMA_CONFIG value so that its upper half (FLOW and NDSIZE) now describes the newly enqueued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMA_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was enqueued too late (that is, the modification of the next-to-last descriptor's DMA_CONFIG element occurred after that element was read into the DMA unit.) In this case, the interrupt handler should write the DMA_CONFIG value appropriate for the last descriptor to the DMA channel's DMA_CONFIG register, increment the completed-descriptor count, and exit.

Again, this system can fail if the system interrupt latencies are large enough to cause any of the channel DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptor interrupts would need to be complex and would need to do several MMR register accesses to ensure robust operation. In such a system environment, a minimal-interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's DMA_CONFIG word should have a FLOW value of zero. If more than one request is received before the DMA-queue-completion interrupt occurs, the non-interrupt code should enqueue later descriptors, forming a waiting portion of queue that is disconnected from the active portion of queue being processed by the DMA unit. In other words, all but the last active descriptors contain FLOW values >= 4 and have no interrupt enable set, while the last active descriptor contains a FLOW of 0 and an interrupt enable bit DI_EN set to 1. Also, all but the last waiting descriptors contain FLOW values >=4 and no interrupt enables set, while the last waiting descriptor contains a FLOW of 0 and an interrupt enable bit set to 1. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler by a single DMA_CONFIG register write.

After enqueuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location, containing the desired DMA_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting.)

It is critical that the software not modify the contents of the active descriptor queue directly, once its processing by the DMA unit has been started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

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

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

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped, that is, if the mailbox from the interrupt software is zero, the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMA_CONFIG value to the channel's DMA_CONFIG register). If the queue is not stopped, however, the non-interrupt software must not write the DMA_CONFIG register (which would cause a DMA error), but instead it should enqueue the descriptor onto the waiting queue and update its mailbox directed to the interrupt handler.

More 2D DMA Examples

Examples of DMA styles supported by flex descriptors include:

- Single linear buffer that stops on completion (FLOW = Stop mode).
- Linear buffer with stride greater than one (X_MODIFY > 1).
- Circular, auto-refreshing buffer that interrupts on each full buffer.
- Similar buffer that interrupts on fractional buffers (for example, $^1/_2$, $^1/_4$) (2D DMA).
- 1D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing { link pointer, 32-bit address }.

- 1D DMA, using a linked list of 5-word descriptors containing { link pointer, 32-bit address, length, config } (ADSP-2191 style).
- 2D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page.
- 2D DMA, using a linked list of 9-word descriptors, specifying everything.

Memory DMA

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

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

Each DMA controller provides four MDMA channels:

- two source channels (for reading from memory)
- two destination channels (for writing to memory)

There are two DMA controllers, each containing two streams. The DMA controllers arbitrate between themselves, with DMA controller 0 taking priority over DMA controller 1. Within each DMA controller, the two streams have the following priorities:

highest priority	memory DMA destination stream D0
	memory DMA source stream S0
	memory DMA destination stream D1
lowest priority	memory DMA source stream S1

Because lower priority values take precedence over higher values, memory DMA stream 0 takes precedence over memory DMA stream 1, unless round-robin scheduling is used. Note it is illegal to program a source stream for memory write and it is illegal to program a destination stream for memory read.

The channels support 8-bit, 16-bit, and 32-bit memory DMA transfers, but both ends of the MDMA transfer must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word-deep 16-bit FIFO. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the EAB and DAB buses to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total count is the same.

To start an MDMA transfer operation, the MMR registers for the source and destination streams are written, each in a manner similar to peripheral DMA. The only constraint is that the DMA_CONFIG register for the source stream must be written before the DMA_CONFIG register for the destination stream. When the destination DMA_CONFIG register is written, MDMA operation starts, after a latency of three SCLK cycles.

First, if either MDMA stream has been selected to use descriptors, the descriptors are fetched from memory. The destination stream descriptors are fetched first. Then, after a latency of 4 SCLK cycles after the last descriptor word is returned from memory (or typically 8 SCLK cycles after the fetch of the last descriptor word, due to memory pipelining), the source MDMA stream begins fetching data from the source buffer. The resulting data is deposited in the MDMA stream 8-location FIFO, and then after a latency of 2 SCLK cycles, the destination MDMA stream begins writing data to the destination memory buffer.

MDMA Bandwidth

If source and destination buffers are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. At a clock ratio of 2:1, for example, DMA typically runs at 2/3 of the system clock rate. At higher clock ratios, full bandwidth is maintained.

If source and destination buffers are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to 3 plus the memory latency at the DMA in SCLKs (typically 7 for internal transfers and 6 for external transfers).

MDMA Priority and Scheduling

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

MemDMA streams on DMA Controller 0 have priority over peripheral DMA streams on DMA Controller 1.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the MDMA_ROUND_ROBIN_PERIOD field in the DMAx_TCPER register (see "Prioritization and Traffic Control" on page 9-62).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA stream 0 takes precedence over MDMA stream 1 whenever stream 0 is ready to perform transfers. Since an MDMA stream is typically capable of transferring data on every available cycle, this could cause MDMA stream 1 traffic to be delayed for an indefinite time until any and all MDMA stream 0 operations are complete. This scheme could be appropriate in systems where low-duration but latency-sensitive data buffers need to be moved immediately, interrupting long-duration, low-priority background transfers.

If the MDMA_ROUND_ROBIN_PERIOD field is set to some nonzero value in the range 1 <= P <= 31, then a round robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to P data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for external-to-internal moves, and each would be allocated approximately equal data bandwidth.

In round robin operation, the MDMA stream selection at any time is either *free* or *locked*. Initially, the selection is free. On any free cycle available to MDMA (when no PDMA accesses take precedence), if either or both MDMA streams request access, the higher-precedence stream is granted (stream 0 in case of conflict), and that stream selection is then locked. The MDMA_ROUND_ROBIN_COUNT counter field in the DMAx_TCCNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of 1, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and on the next cycle the stream selection unlocks and becomes free again.

If round robin operation is used when only one MDMA stream is active, one idle cycle will occur for each P MDMA data cycles, slightly lowering bandwidth by a factor of 1/(P+1). If both MDMA streams are used, however, memory DMA can operate continuously with zero additional overhead for alternation of streams (other than overhead cycles normally associated with reversal of read/write direction to memory, for example).

By selection of various round-robin period values P which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

DMA Controller Errors (Aborts)

The two DMA controllers each have the ability to generate a DMA controller error.

DMA controller errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

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

The DMA_ERROR interrupt handler must perform the following for each channel:

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

The following error conditions are detected by the DMA hardware.

- A disallowed register write occurred while the channel was running. Only the DMA_CONFIG and IRQ_STATUS registers can be written when DMA_RUN = 1.

- An address alignment error occurred during any memory access. For example, DMA_CONFIG register WDSIZE = 1 (16 bit) but address LSB is not equal to b#0, or WDSIZE = 2 (32 bit) but two address LSBs are not equal to b#00.
- A memory space transition was attempted (internal to external or vice versa).
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signalled by the external memory interface).

They result in a DMA abort interrupt and the configuration register contains the following invalid values.

- Incorrect WDSIZE value (WDSIZE=11)
- Bit 15 not set to 0
- Incorrect FLOW value (FLOW=2, 3, or 5)
- NDSIZE value does not agree with FLOW. See Table 9-11.

Table 9-11. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	0 < NDSIZE <= 7	Descriptor array, no descriptor pointer fetched.
6	0 < NDSIZE <= 8	Descriptor list, small descriptor pointer fetched.
7	0 < NDSIZE <= 9	Descriptor list, large descriptor pointer fetched.

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signalled for the following situations.

- DMA_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMA_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMA_CONFIG word size (WDSIZE) is not supported by the mapped peripheral.
- DMA_CONFIG word size in source and destination of the MDMA stream are not equal.
- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2D DMA, X_COUNT = 1.

DMA Performance: Prioritization and Optimization

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

A key feature of the DMA architecture is the separation of the activity on the peripheral DMA bus (the DAB bus) from the activity on the buses between the DMA and memory (the DCB and DEB buses). Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low

latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

This allows the peripheral DMA channels to have a maximum transfer rate of one 16-bit word per two system clocks per channel in either direction. The MDMA channels have a maximum transfer rate of 1 16-bit word per 1 system clock, per channel.

When all DMA channel traffic is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of 1 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of 1 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of 1 16-bit transfer per system clock.

Some considerations which limit the actual performance are:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example, for accessing the same L1 bank, for opening/closing SDRAM pages, or while filling cache lines.
- Each direction change from RX to TX on the DAB bus imposes a 1-clock delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.

- MMR accesses to registers other than DMAx_CONFIG, DMAx_IRQ_STATUS, or DMAx_PERIPHERAL_MAP will stall all DMA activity for 1 cycle per 16-bit word transferred. In contrast, MMR accesses to the control/register registers do not cause stalls or wait states.
- Reads from registers other than control/register registers use one PAB bus wait state, delaying the core for several core clocks.
- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMA_EN changes from 0 to 1 or when the RESTART bit is set to 1 in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

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

Prioritization and Traffic Control

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the DMAx_PERIPHERAL_MAP registers. The

memory DMA streams are always lower priority than the peripherals on the same DMA controller, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers. By default, when more than one MDMA stream is enabled and ready, only the highest-priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, however, the MDMA_ROUND_ROBIN_PERIOD may be programmed to select each stream in turn for a fixed number of transfers.

In the processor DMA, there are two completely separate but simultaneous prioritization processes: the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see "Urgent DMA Transfers" on page 9-67.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same-direction transfers together. The DMA block provides a traffic control mechanism controlled by the DMACx_TC_PER and DMACx_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention, and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters.

In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMACx_TC_CNT register.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out, or until traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite-direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower-priority channel 5 is going "with traffic" and higher-priority channel 3 is going "against traffic," then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If on the next cycle the only channels requesting DAB transfers were all going against traffic, (channels 3 and 6), then their effective priorities become 19 and 22. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required by the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the DMAx_TC_PER register to 0x0000.

DMA Traffic Control Counter Period (DMACx_TC_PER) Register and DMA Traffic Control Counter (DMACx_TC_CNT) Register

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMACx_TC_PER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle, then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMACx_TC_PER is written, or whenever the DAB bus changes direction or becomes idle, then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DAB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DAB access is preferred, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever DMACx_TC_PER is written, or whenever the DEB bus changes direction or becomes idle, then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DEB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DEB access is preferred, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMACx_TC_PER is written, or whenever the DCB bus changes direction or becomes idle, then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DCB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DCB access is preferred, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from DCB_TRAFFIC_PERIOD to begin a new burst.

DMA Traffic Control Counter Period Registers (DMACx_TC_PER)

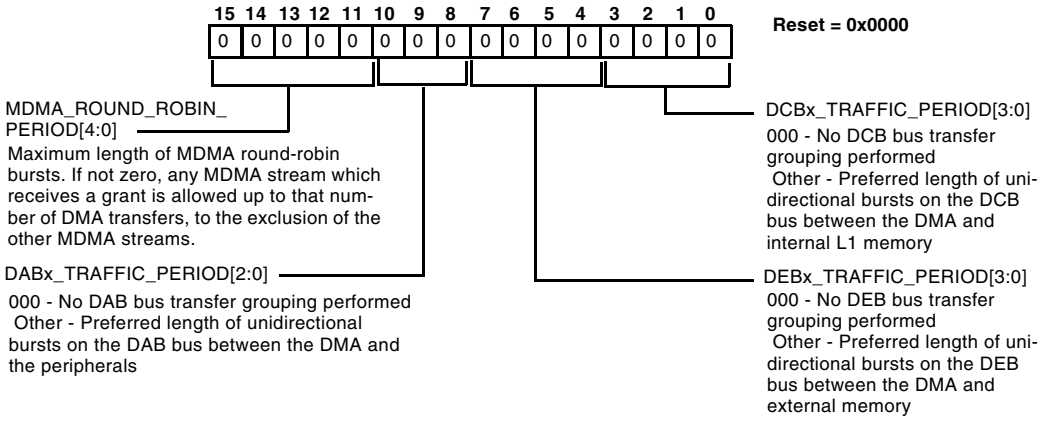


Figure 9-18. DMA Traffic Control Counter Period Register

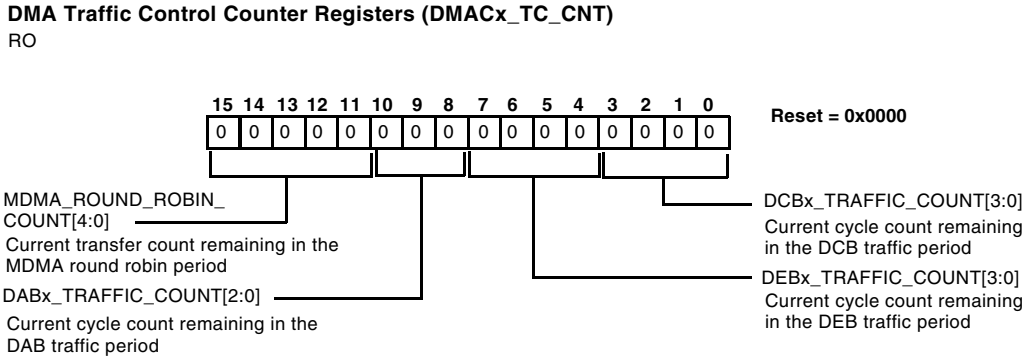


Figure 9-19. DMA Traffic Control Counter Register

Urgent DMA Transfers

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals is not too large a fraction of the total, then all peripherals' requests should be granted as required.

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

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

A DMA channel's request for memory service is defined as urgent if both the channel's FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and the peripheral is asserting its DMA request line.

Descriptor fetches may be urgent, if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral. DMA requests from an MDMA channel are never urgent.

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

1. All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.

2. The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external), and so are all prior incomplete memory transfers ahead of it in that memory system. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

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

10 SPI COMPATIBLE PORT CONTROLLERS

The ADSP-BF539 processor has three serial peripheral interface (SPI) ports that provide an I/O interface to a wide variety of SPI compatible peripheral devices.

With a range of configurable options, SPI ports (`SPI0`, `SPI1`, `SPI2`) provide glueless hardware interface to other SPI compatible devices. An SPI is a four-wire interface consisting of two data pins, a device select pin, and a clock pin. The SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI compatible peripheral implementation also supports programmable baud rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Typical SPI compatible peripheral devices that can be used to interface to the SPI compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays

- Shift registers
- FPGAs with SPI emulation

The SPI is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, four-wire interface consisting of two data pins (MOSIx and MISOx), one device select pin ($\overline{\text{SPIxSS}}$), and a gated clock pin (SCKx). With the two data pins, it allows for full-duplex operation to other SPI compatible devices. The SPI also includes programmable baud rates, clock phase, and clock polarity.

The SPI can operate in a multimaster environment by interfacing with several other devices, acting as either a master device or a slave device. In a multimaster environment, the SPI interface uses open drain outputs to avoid data bus contention.

Figure 10-1 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCKx 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 SCKx synchronizes the shifting and sampling of the data on the two serial data pins.

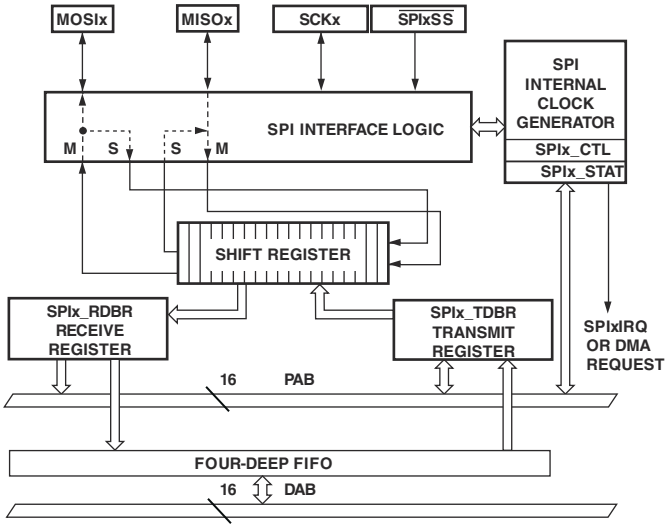


Figure 10-1. SPI Block Diagram

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 (SPIxSS). The other SPI device acts as the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode on SPI0, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected via their SPI ports, all MOSIx pins are connected together, all

MISOx pins are connected together, and all SCKx pins are connected together.

For a multislave environment, the processor can make use of seven programmable flags, PF1–PF7, that are dedicated SPI0 slave select signals for the SPI0 slave devices. For SPI1 and SPI2, the processor uses a single GPIO pin for a single slave-select output each. SPI1 uses the PD4 pin, and SPI2 uses the PD9 pin in this capacity.

At reset, the SPI is disabled and configured as a slave.

Interface Signals

The following section discusses the SPI signals.

Serial Peripheral Interface Clock Signals (SCKx)

The SCKx signal is the SPI clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of baud rates. The SCKx signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCKx is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the Serial Peripheral Slave Select Input ($\overline{\text{SPIxSS}}$) is driven inactive (high).

The SCKx is used to shift out and shift in the data driven on the MISOx and MOSIx lines. Clock polarity and clock phase relative to data are programmable in the SPI control register (SPIx_CTL) and define the transfer format (see "SPI Transfer Formats" on page 10-21).

The SCK0 signal is dedicated. The SCK1 and SCK2 signals are GPIO pins PD2 and PD7, respectively. Be sure to set the GPIO_D_CNFG register for peripheral use. See Chapter 15, "General Purpose Input/Output Ports".

Serial Peripheral Interface Slave Select Input Signals (SPIxSS)

The 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 multi-master mode, if the SPIxSS input signal of a master is asserted (driven low), and the PSSE bit in the SPIx_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The SPI0SS signal is the same pin as the PF0 pin. Be careful to not use PF0 as an output if intended to serve as the SPI0SS. The SPI1SS and SPI2SS signals are GPIO pins (PD3 and PD8, respectively). Be sure to set the GPIO_D_CNFG register for peripheral use. See Chapter 15, "General Purpose Input/Output Ports".

Master Out Slave In (MOSIx)

The MOSIx pin is the master-out-slave-in pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSIx pin becomes a data transmit (output) pin, transmitting output data. If the processor is configured as a slave, the MOSIx pin becomes a data receive (input) pin, receiving input data. In an SPI interconnection, the data is shifted out from the MOSIx output pin of the master and shifted into the MOSIx input(s) of the slave(s).

The MOSI0 pin is dedicated, and the MOSI1 and MOSI2 pins are GPIO pins (PD0 and PD5, respectively). Be sure to set the GPIO_D_CNFG register for peripheral use. See Chapter 15, "General Purpose Input/Output Ports".

Master In Slave Out (MISOx)

The MISOx pin is the master-in-slave-out pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MISOx pin becomes a data receive (input) pin, receiving input data. If the processor is configured as a slave, the MISOx pin becomes a data transmit (output) pin, transmitting output data. In an SPI interconnection, the data is shifted out from the MISOx output pin of the slave and shifted into the MISOx input pin of the master. The following important points should be noted.

- The MISO0 pin is dedicated, and the MISO1 and MISO2 pins are GPIO pins (PD1 and PD6, respectively). Be sure to set the GPIO_D_CNFG register for peripheral use. See Chapter 15, "General Purpose Input/Output Ports".
- In a multi-slave environment, only one slave is allowed to transmit data at any given time.
- The processor can be booted via its SPI0 interface to allow user application code and data to be downloaded before runtime.

The SPI configuration example in Figure 10-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

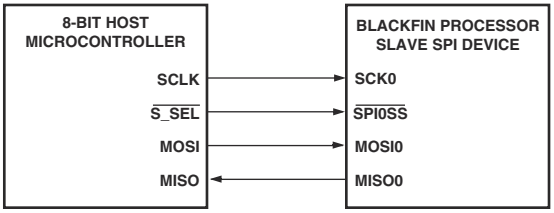


Figure 10-2. ADSP-BF539 Blackfin Processor as a Slave SPI Device

Interrupt Output

Each SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the transfer Initiation mode bit field (TIMOD) in the SPI control register. In DMA mode (TIMOD = 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 (SPIx_CTL) Register" on page 10-9.

SPI Registers

The SPI peripherals include a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPIx_BAUD, SPIx_CTL, SPIx_FLG, and SPIx_STAT. Two registers are used for buffering receive and transmit data: SPIx_RDBR and SPIx_TDBR. For information about DMA-related registers, see Chapter 9, "Direct Memory Access". The shift register, SFDR, is internal to the SPI module and is not directly accessible.

See "Error Signals and Flags" on page 10-30 for more information about how the bits in these registers are used to signal errors and other conditions. See "Register Functions" on page 10-21 for more information about SPI register and bit functions.

SPI BAUD Rate (SPIx_BAUD) Register

The SPI baud rate register (SPIx_BAUD) is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

SCKx frequency = (peripheral clock frequency SCLK)/(2 x 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.

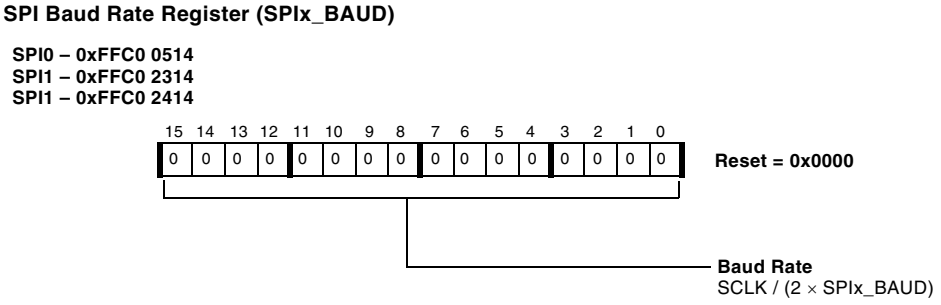


Figure 10-3. SPI Baud Rate Registers

Table 10-1 lists several possible baud rate values for SPIx_BAUD.

Table 10-1. SPI Master Baud Rate Example

SPIx_BAUD Decimal Value	SPI Clock (SCKx) 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

SPI Control (SPIx_CTL) Register

The SPI control register (SPIx_CTL) is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPIx_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to b#00, a SPI port transaction is begun when the receive buffer is read. Data from the first read needs to be discarded since the read is needed to initiate the first SPI port transaction. When set to 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 MISOx pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the status register is set. See “Mode Fault Error (MODF)” on page 10-30.

Figure 10-4 provides the bit descriptions for SPIx_CTL.

SPI Flag (SPIx_FLG) Register

If the SPI is enabled as a master, the SPI uses the SPI flag register (SPIx_FLG) to enable up to seven general-purpose programmable flag pins to be used as individual slave select lines. In Slave mode, the SPIx_FLG bits have no effect, and each SPI uses the $\overline{\text{SPIxSS}}$ input as a slave select.

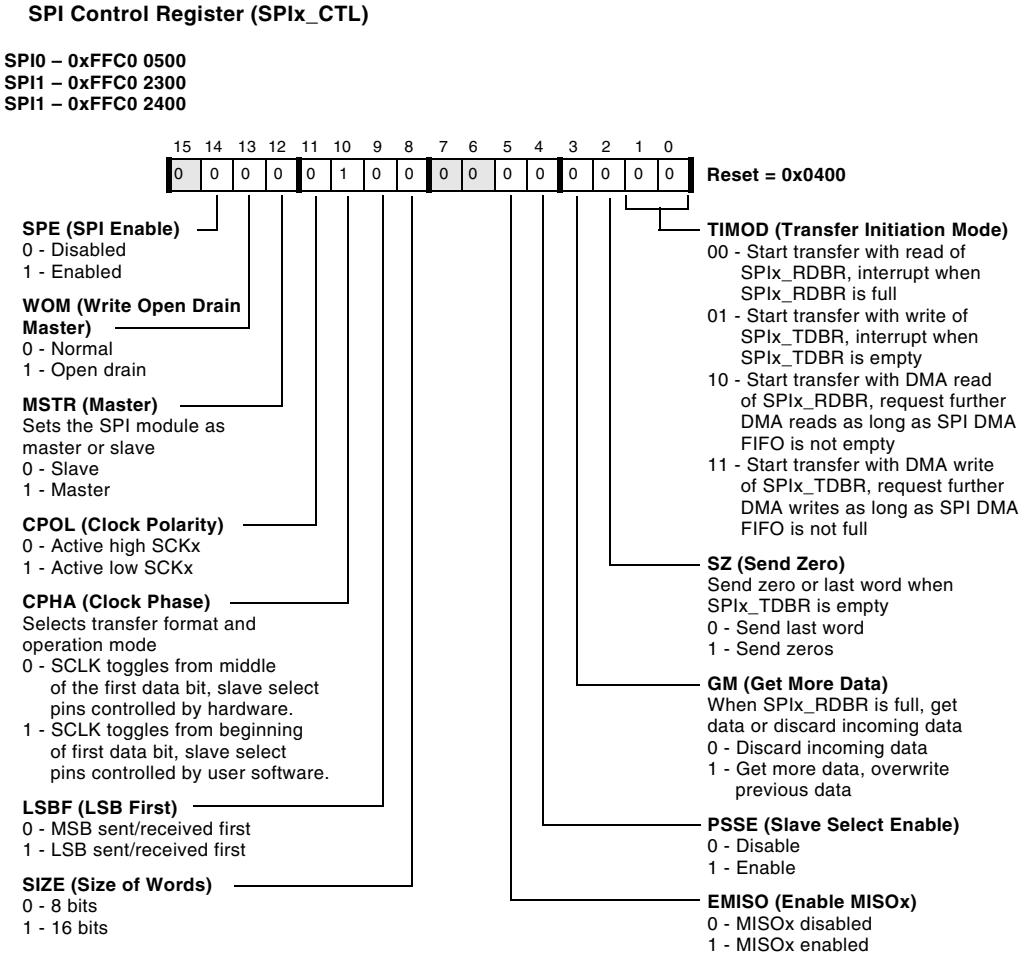


Figure 10-4. SPI Control Register

Figure 10-5 shows the SPIO_FLG register diagram. For details specific to the SPI1 and SPI2 ports, see "Special Considerations for SPI1 and SPI2 Slave Control" on page 10-16.

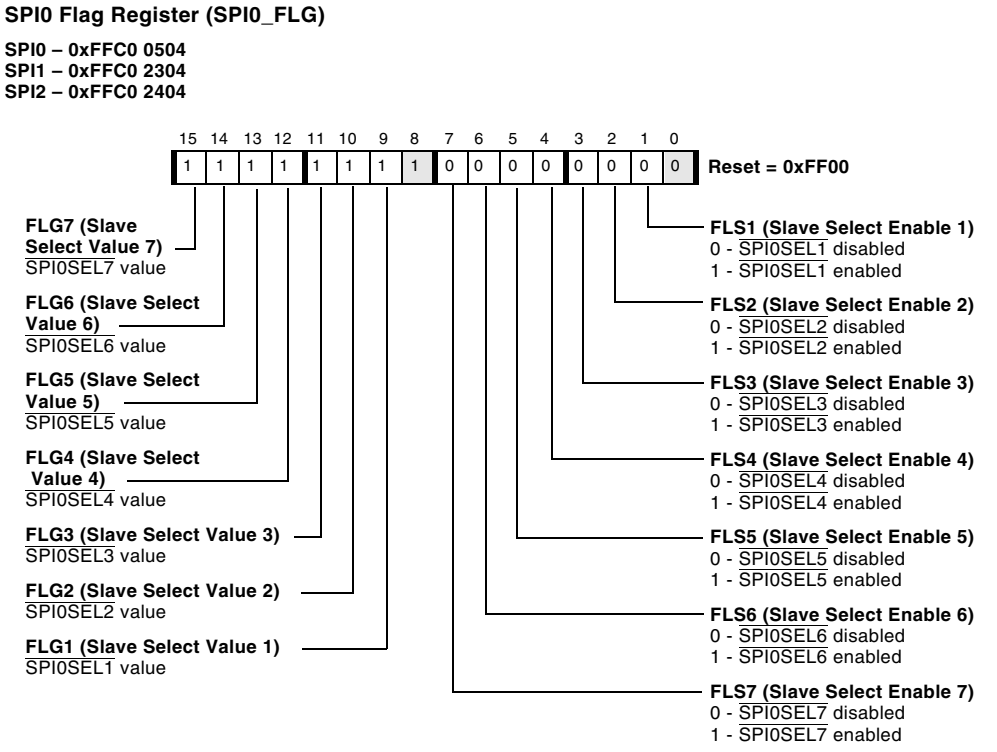


Figure 10-5. SPI0 Flag Register

The SPI0_FLG register consists of two sets of bits that function as follows.

- slave select enable (FLSx) bits

  Each FLSx bit corresponds to a programmable flag (PFx) pin. When a FLSx bit is set, the corresponding PFx pin is driven as a slave select. For example, if FLS1 is set in SPI0_FLG, PF1 is driven as a slave select (SPI0SEL1). Table 10-2 shows the association of the FLSx bits and the corresponding PFx pins.

If the FLSx bit is not set, the general-purpose programmable flag registers (FIO_DIR and others) configure and control the corresponding PFx pin for SPI0.

- Slave select value (FLGx) bits
- When a PFx pin is configured as a slave select output for SPI0, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPIO_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, to drive PF3 as a slave select, FLS3 in SPIO_FLG must be set. Clearing FLG3 in SPIO_FLG drives PF3 low; setting FLG3 drives PF3 high. The PF3 pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, PF3 remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use PF3 as a slave select pin, it is only necessary to set the FLS3 bit in SPIx_FLG. It is not necessary to write to the FLG3 bit, because the SPI hardware automatically drives the PF3 pin.

Table 10-2. SPI0_FLG Bit Mapping to PFx Pins

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	$\overline{\texttt{SPI0SEL1}}$ Enable	PF1	0
2	FLS2	$\overline{\texttt{SPI0SEL2}}$ Enable	PF2	0
3	FLS3	$\overline{\texttt{SPI0SEL3}}$ Enable	PF3	0
4	FLS4	$\overline{\texttt{SPI0SEL4}}$ Enable	PF4	0
5	FLS5	$\overline{\texttt{SPI0SEL5}}$ Enable	PF5	0
6	FLS6	$\overline{\texttt{SPI0SEL6}}$ Enable	PF6	0
7	FLS7	$\overline{\texttt{SPI0SEL7}}$ Enable	PF7	0
8	Reserved			1
9	FLG1	$\overline{\texttt{SPI0SEL1}}$ Value	PF1	1
10	FLG2	$\overline{\texttt{SPI0SEL2}}$ Value	PF2	1
11	FLG3	$\overline{\texttt{SPI0SEL3}}$ Value	PF3	1
12	FLG4	$\overline{\texttt{SPI0SEL4}}$ Value	PF4	1
13	FLG5	$\overline{\texttt{SPI0SEL5}}$ Value	PF5	1
14	FLG6	$\overline{\texttt{SPI0SEL6}}$ Value	PF6	1
15	FLG7	$\overline{\texttt{SPI0SEL7}}$ Value	PF7	1

Slave Select Inputs

If the SPI is in slave mode, $\overline{\texttt{SPIxSS}}$ acts as the slave select input. When enabled as a master, $\overline{\texttt{SPIxSS}}$ 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 $\overline{\texttt{SPIxSS}}$ input is the master mode error input. Otherwise, $\overline{\texttt{SPIxSS}}$ is ignored.

Use of FLS Bits in SPI0_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI0_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system

including a processor master, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI0 is the master. The seven flag pins (PF1-PF7) on the processor master can be connected to each of the slave SPI device's $\overline{\text{SPIxSS}}$ slave-select input pins. In this configuration, the FLSx bits in SPI0_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected via SPI ports can be other processors.
3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 10-6 shows one processor as a master with three processors (or other SPI compatible devices) as slaves.

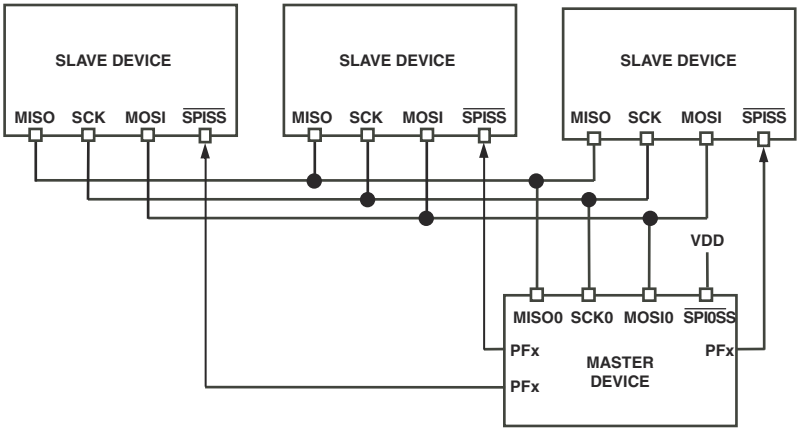


Figure 10-6. Single-Master, Multiple-Slave Configuration

Special Considerations for SPI1 and SPI2 Slave Control

When configured as a master, all functionality and control for the slave-select outputs for SPI1 and SPI2 are exactly as described for SPI0 above. However, since SPI1 and SPI2 can only control one slave-select output signal, the only functional bits in the relevant `SPIx_FLG` registers are the `FLS1`/`FLG1` bits. The rest of the bits in these registers are reserved. For SPI1, modifying these bit locations affects the `PD4` pin. For SPI2, modifying these bit locations affects the `PD9` pin. When configured as a slave, SPI1 and SPI2 utilize the `PD3` and `PD8` pins, respectively. Be sure to set the `GPIO_D_CNFG` register for peripheral use. See "General Purpose Input/Output Ports" in Chapter 15, General Purpose Input/Output Ports.

SPI Status (SPIx_STAT) Register

The SPI status register (SPIx_STAT) is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

Some of the bits in SPIx_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a 1 to the desired bit position of SPIx_STAT. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPIx_STAT to clear the TXE error condition. This allows the user to read SPIx_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

SPI Status Register (SPIx_STAT)

SPI0 – 0xFFC0 0508
SPI1 – 0xFFC0 2308
SPI2 – 0xFFC0 2408

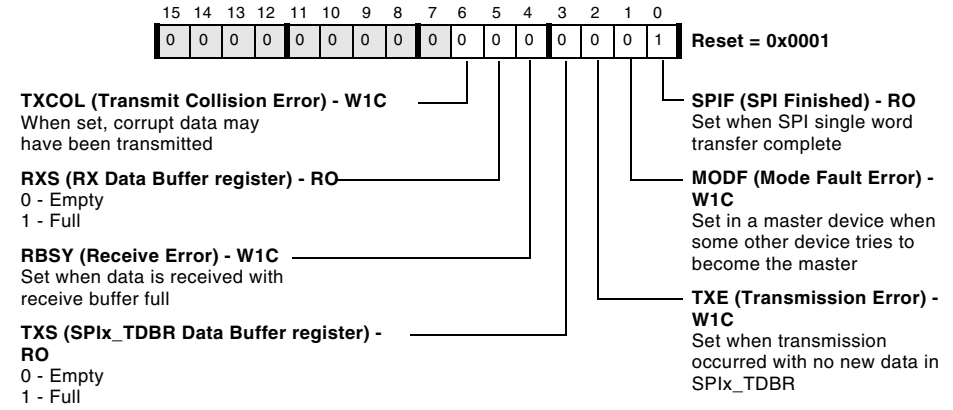


Figure 10-7. SPIx Status Register

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPIx_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently is set, the last word has been transferred.

SPI Transmit Data Buffer (SPIx_TDBR) Register

The SPI transmit data buffer register (SPIx_TDBR) is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPIx_TDBR is loaded into the shift data register (SFDR). A read of SPIx_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPIx_TDBR should not occur in this mode because this data overwrites the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPIx_TDBR are repeatedly transmitted. A write to SPIx_TDBR is permitted in this mode, and this data is transmitted.

If the send zeros control bit (SZ in the SPIx_CTL register) is set, SPIx_TDBR may be reset to 0 under certain circumstances.

If multiple writes to SPIx_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPIx_TDBR are transmitted. Multiple writes to SPIx_TDBR are possible, but not recommended.

SPI Transmit Data Buffer Register (SPIx_TDBR)

SPI0 – 0xFFC0 050C
SPI1 – 0xFFC0 230C
SPI2 – 0xFFC0 240C

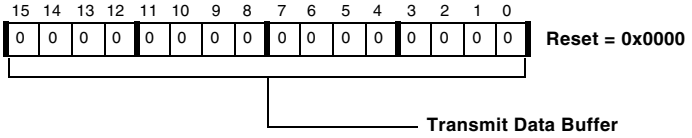


Figure 10-8. SPIx Transmit Data Buffer Register

SPI Receive Data Buffer Register (SPIx_RDBR)
RO

SPI0 – 0xFFC0 0510
SPI1 – 0xFFC0 2310
SPI2 – 0xFFC0 2410

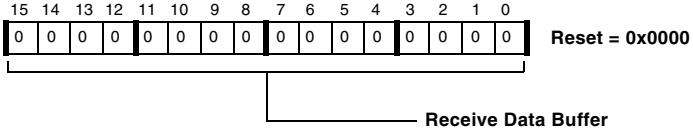


Figure 10-9. SPIx Receive Data Buffer Register

SPI Receive Data Buffer (SPIx_RDBR) Register

The SPI receive data buffer register (SPIx_RDBR) is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded

into SPIx_RDBR. During a DMA receive operation, the data in SPIx_RDBR is automatically read by the DMA. When SPIx_RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD = b#00).

SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register

The SPI RDBR shadow register (SPIx_SHADOW), has been provided for use in debugging software. This register is at a different address than the receive data buffer, SPIx_RDBR, but its contents are identical to that of SPIx_RDBR. When a software read of SPIx_RDBR occurs, the RXS bit in SPIx_STAT is cleared and an SPI transfer may be initiated (if TIMOD = b#00 in SPIx_CTL). No such hardware action occurs when the SPIx_SHADOW register is read. The SPIx_SHADOW register is read-only.

SPI RDBR Shadow Register (SPIx_SHADOW)
RO

SPI0 – 0xFFC0 0518
SPI1 – 0xFFC0 2318
SPI2 – 0xFFC0 2418

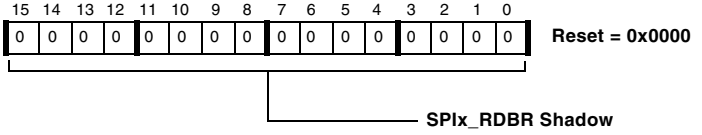


Figure 10-10. SPIx RDBR Shadow Register

Register Functions

Table 10-3 summarizes the functions of the SPI registers.

Table 10-3. SPI Register Mapping

Register Name	Function	Notes
SPIx_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPIx_FLG	SPI port flag	Bits 0 and 8 are reserved in SPI0_FLG. All bits except FLS1 and FLG1 are reserved in SPI1_FLG and SPI2_FLG.
SPIx_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPIx_CTL
SPIx_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPIx_CTL)
SPIx_RDBR	SPI port receive data buffer	When register is read, hardware events are triggered
SPIx_BAUD	SPI port baud control	Value of 0 or 1 disables the serial clock
SPIx_SHADOW	SPI port data	Register has the same contents as SPIx_RDBR, but no action is taken when it is read

SPI Transfer Formats

The SPI supports four different combinations of serial clock phase and polarity (SPI modes 0-3). These combinations are selected using the CPOL and CPHA bits in SPIx_CTL, as shown in Figure 10-11.

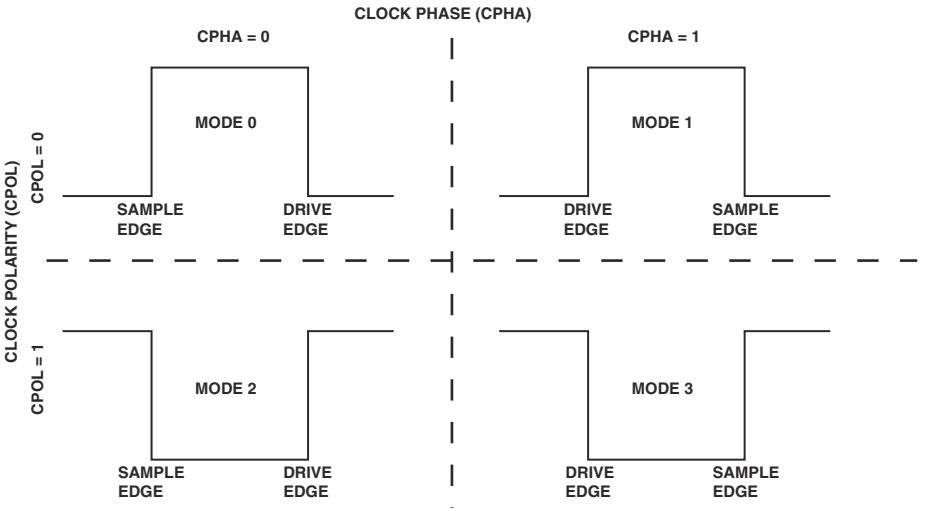


Figure 10-11. SPI Modes of Operation

The figures "SPI Transfer Protocol for CPHA = 0" on page 10-24 and "SPI Transfer Protocol for CPHA = 1" on page 10-24 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SCKx—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCKx, MISOx, and MOSIx pins are directly connected between the master and the slave. The MISOx signal is the output from the slave (slave transmission), and the MOSIx signal is the output from the master (master transmission). The SCKx signal is generated by the master, and the $\overline{\text{SPIxSS}}$ signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with the Most Significant Bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of

SPIx_CTL is allowed. For example, a 16-bit transfer with the Least Significant Bit (LSB) first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When CPHA = 0, the slave select line outputs, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1,select line outputs $\overline{\text{SPIxSS}}$ may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software via manipulation of SPIx_FLG.

Figure 10-12 shows the SPI transfer protocol for CPHA = 0. Note SCKx starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

Figure 10-13 shows the SPI transfer protocol for CPHA = 1. Note SCKx starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

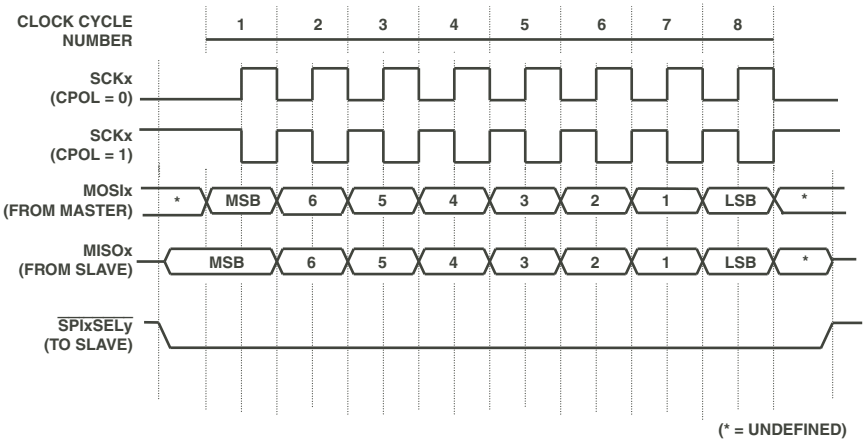


Figure 10-12. SPI Transfer Protocol for CPHA = 0

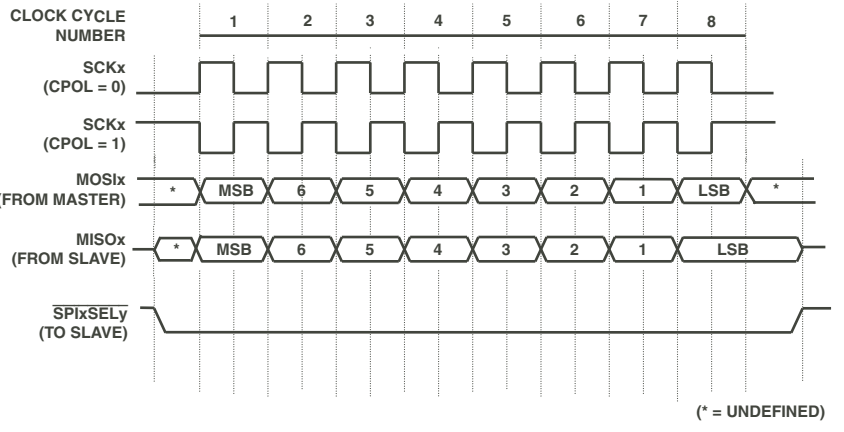


Figure 10-13. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in a single master as well as multimaster environment. The MOSIx, MISOx, and the SCKx signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the Broadcast mode has been selected. In Broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in Transmit mode driving the MISOx line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSIx and MISOx) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSIx and MISOx pins when this option is selected.

The WOM bit controls this option. When WOM is set and the SPI is configured as a master, the MOSIx pin is three-stated when the data driven out on MOSIx is a logic high. The MOSIx pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISOx pin is three-stated if the data driven out on MISOx is a logic high.

Clock Signals

The SCKx signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value of SPIx_BAUD. For slave devices, the value in SPIx_BAUD is ignored. When the SPI device is a master, SCKx is an output signal. When the SPI is a slave, SCKx is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCKx signal is used to shift out and shift in the data driven onto the MISOx and MOSIx lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and clock phase relative to data are programmable into SPIx_CTL and define the transfer format (see Figure 10-11 on page 10-22).

Master Mode Operation

When the SPI0 is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to SPI0_FLG, setting one or more of the SPI Flag Select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.
2. The core writes to the SPI0_BAUD and SPI0_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.
3. If CPHA = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI0_FLG.

4. The TIMOD bits in SPIO_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 (SPIO_TDBR) or a data read of the receive data buffer (SPIO_RDBR).
5. The SPI then generates the programmed clock pulses on SCKO and simultaneously shifts data out of MOSIO and shifts data in from MISOO. Before a shift, the shift register is loaded with the contents of the SPIO_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPIO_RDBR.
6. With each new Transfer Initiate command, the SPI continues to send and receive words, according to the SPI Transfer Initiate mode.

For SPI1 and SPI2, the SPI pins must not be enabled for GPIO. See Chapter 15, "General Purpose Input/Output Ports". 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 MOSIx 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 MISOx 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 10-4.

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 10-4. 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 SPI for DMA mode. Individual word transfers begin with a DMA read of SPIx_RDBR, and last transfer completed.	Request DMA reads as long as SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPIx_TDBR, and last transfer completed.	Request DMA writes as long as SPI DMA FIFO is not full.

Slave Mode Operation

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the $\overline{\text{SPIxSS}}$ select signal to the active state (low), or by the first active edge of the clock (SCKx), depending on the state of CPHA.

These steps illustrate SPI operation in the slave mode:

1. The core writes to SPIx_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.
2. To prepare for the data transfer, the core writes data to be transmitted into SPIx_TDBR.
3. Once the $\overline{\text{SPIxSS}}$ falling edge is detected, the slave starts shifting data out on MISOx and in from MOSIx on SCKx edges, depending on the states of CPHA and CPOL.
4. Reception/transmission continues until $\overline{\text{SPIxSS}}$ is released or until the slave has received the proper number of clock cycles.
5. The slave device continues to receive/transmit with each new falling edge transition on $\overline{\text{SPIxSS}}$ and/or SCKx clock edge.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPIx_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MISOx pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSIx pin, overwriting the older data in SPIx_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPIx_RDBR is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 10-5 are necessary to prepare the device for a new transfer.

Table 10-5. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and Receive	Interrupt active when receive buffer is full. Read of SPIx_RDBR clears interrupt.
b#01	Transmit and Receive	interrupt active when transmit buffer is empty. Writing to SPIx_TDBR clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Error Signals and Flags

The status of a device is indicated by the SPIx_STAT register. See "SPI Status (SPIx_STAT) Register" on page 10-17 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 (MOSIx, MISOx, and SCKx) are disabled. However, the slave select output pins revert to being controlled by the programmable flag or GPIO 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 programmable flag or GPIO registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the PFx pins used as slave selects as inputs. Programs can do this by configuring the direction of the slave-select pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as programmable pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPIx_STAT when all the conditions of transmission are met, and there is no new data in SPIx_TDBR (SPIx_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPIx_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPIx_STAT register when a new transfer is completed, but before the previous data can be read from SPIx_RDBR. The state of the GM bit in the SPIx_CTL register determines whether SPIx_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPIx_STAT when a write to SPIx_TDBR coincides with the load of the shift register. The write to SPIx_TDBR can be via software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPIx_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, whether the CPHA mode is selected, and whether the Transfer Initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPIx_TDBR is written to or SPIx_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCKx signal remains inactive for the first half of the first cycle of SCKx. For a slave with CPHA = 0, the transfer starts as soon as the $\overline{\text{SPIxSS}}$ input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCKx for both slave and master devices. For a master device, a transfer is considered finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCKx.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPIx_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCKx. For a slave SPI, RXS is set shortly after the last SCKx edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPIx_RDBR is full (TIMOD = b#00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF is cleared shortly after the start of a transfer ($\overline{\text{SPIxSS}}$ going low for CPHA = 0, first active edge of SCKx on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a transfer (either by writing the SPIx_TDBR or reading the SPIx_RDBR, depending on TIMOD), and is set one-half SCKx period after the last SCKx edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPIx_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPIx_RDBR, because of the latency. Therefore, for SPIx_BAUD = 2 or SPIx_BAUD = 3, RXS must be set before SPIF to read SPIx_RDBR. For larger SPIx_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the 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 has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in TIMOD = 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.

DMA

The SPI ports also can use direct memory Access (DMA). For more information on DMA, see “DMA and Memory DMA MMRs” on page 9-3.

DMA Functionality

Each SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data is essentially ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DMA Access Bus (DAB).

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPIx_STAT register until it goes low for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred.

The four-word FIFO is cleared when the SPI port is disabled.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI0 interface operates as follows.

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA Channel and to configure the necessary work units, access direction, word count, and so on. For more information, see "DMA and Memory DMA MMRs" on page 9-3.
2. The processor core writes to the SPI0_FLG register, setting one or more of the SPI flag select bits (FLSx).
3. The processor core writes to the SPI0_BAUD and SPI0_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "Receive with DMA" (TIMOD = b#10) or "Transmit with DMA" (TIMOD = b#11) mode.

4. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPIO_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPIO_TDBR register, it initiates a transfer on the SPI link.

5. The SPI then generates the programmed clock pulses on SCKO and simultaneously shifts data out of MOSIO and shifts data in from MISOO. For receive transfers, the value in the shift register is loaded into the SPIO_RDBR register at the end of the transfer. For transmit transfers, the value in the SPIO_TDBR register is loaded into the shift register at the start of the transfer.

6. In Receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA Word Count register transitions from 1 to 0. The SPI continues receiving words until SPI DMA mode is disabled.

   For SPI1 and SPI2, the SPI pins must not be enabled for GPIO. See Chapter 15, “General Purpose Input/Output Ports”. In Transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA Word Count register transitions from 1 to 0. The SPI continues transmitting words until the SPI DMA FIFO is empty.

For receive DMA operations, if the DMA engine is unable to keep up with the receive 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 MISOx 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 MOSIx 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 does 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 = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 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 SCKx, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA Channel and configure the necessary work units, access direction, word count, and so on. For more information, see "DMA and Memory DMA MMRs" on page 9-3.
2. The processor core writes to the SPIx_CTL register to define the mode of the serial link to be the same as the mode setup in the SPI master. The TIMOD field is configured to select either *receive with DMA* (TIMOD = b#10) or *transmit with DMA* (TIMOD = b#11) mode.
3. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCKx edges. The value in the shift register is loaded into the SPIx_RDBR register at the end of the transfer. As the SPI reads data from the SPIx_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

For SPI1 and SPI2, the SPI pins must not be enabled for GPIO. See Chapter 15, "General Purpose Input/Output Ports".

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPIx_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCKx edges. The value in the SPIx_TDBR register is loaded into the shift register at the start of the transfer.

4. In Receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA Word Count register transitions from 1 to 0. The SPI slave continues receiving words on SCKx sampling edges as long as the slave select input is active.

   In Transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA Word Count register transitions from 1 to 0. The SPI slave continues transmitting words on SCKx drive edges as long as the slave select input is active.

For receive DMA operations, if the DMA engine is unable to keep up with the receive 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 MOSIx pin, overwriting the older data in the SPIx_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPIx_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits 0s on the MISOx pin. If SZ = 0, it repeatedly transmits the contents of the SPIx_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits 0s on the MISOx pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in Transmit DMA mode, including the data in the SPIx_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPIx_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPIx_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPIx_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 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).

Timing

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCKx period. See Figure 10-14. The minimum time between successive word transfers (T4) is two SCKx periods. This is measured from the last active edge of SCKx of one word to the first active edge of SCKx of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

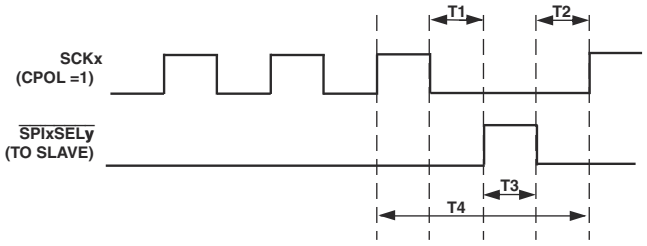


Figure 10-14. SPI Timing

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SCKx period. In this case, T1 and T2 are each always be equal to one-half the SCKx period.

11 PARALLEL PERIPHERAL INTERFACE

The Parallel Peripheral Interface (PPI) is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin, three multiplexed frame sync pins, and four dedicated data pins. Up to 12 additional data pins are available by reconfiguring the PF pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

The PPI_CLK pin can accept an external clock input up to SCLK/2. It cannot source a clock internally. Table 11-1 shows the pin interface for the PPI.

If a programmable flag pin is configured for PPI use, its bit position in programmable flag MMRs will read back as 0.

Table 11-1. PPI Pins

Signal Name	Function	Direction	Alternate Function
PPI15	Data	Bidirectional	PF4, $\overline{\text{SPIOSEL4}}$
PPI14	Data	Bidirectional	PF5, $\overline{\text{SPIOSEL5}}$
PPI13	Data	Bidirectional	PF6, $\overline{\text{SPIOSEL6}}$
PPI12	Data	Bidirectional	PF7, $\overline{\text{SPIOSEL7}}$
PPI11	Data	Bidirectional	PF8
PPI10	Data	Bidirectional	PF9
PPI9	Data	Bidirectional	PF10
PPI8	Data	Bidirectional	PF11

Table 11-1. PPI Pins (Cont'd)

Signal Name	Function	Direction	Alternate Function
PPI7	Data	Bidirectional	PF12
PPI6	Data	Bidirectional	PF13
PPI5	Data	Bidirectional	PF14
PPI4	Data	Bidirectional	PF15
PPI3	Data	Bidirectional	N/A
PPI2	Data	Bidirectional	N/A
PPI1	Data	Bidirectional	N/A
PPI0	Data	Bidirectional	N/A
PPI_FS3	Frame Sync3/Field	Bidirectional	PF3, $\overline{\texttt{SPIOSEL3}}$
PPI_FS2	Frame Sync2/VSYNC	Bidirectional	Timer 2
PPI_FS1	Frame Sync1/HSYNC	Bidirectional	Timer 1
PPI_CLK	Up to SCLK/2	Input Clock	N/A

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (PPI_CONTROL), the PPI status register (PPI_STATUS), the Delay Count register (PPI_DELAY), the Transfer Count register (PPI_COUNT), and the Lines Per Frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI_CONTROL Register

The PPI control register (PPI_CONTROL) configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 11-1 on page -4 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any PF pins that are unused by the PPI as a result of the DLEN setting are free to be used in their normal PF capacity.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

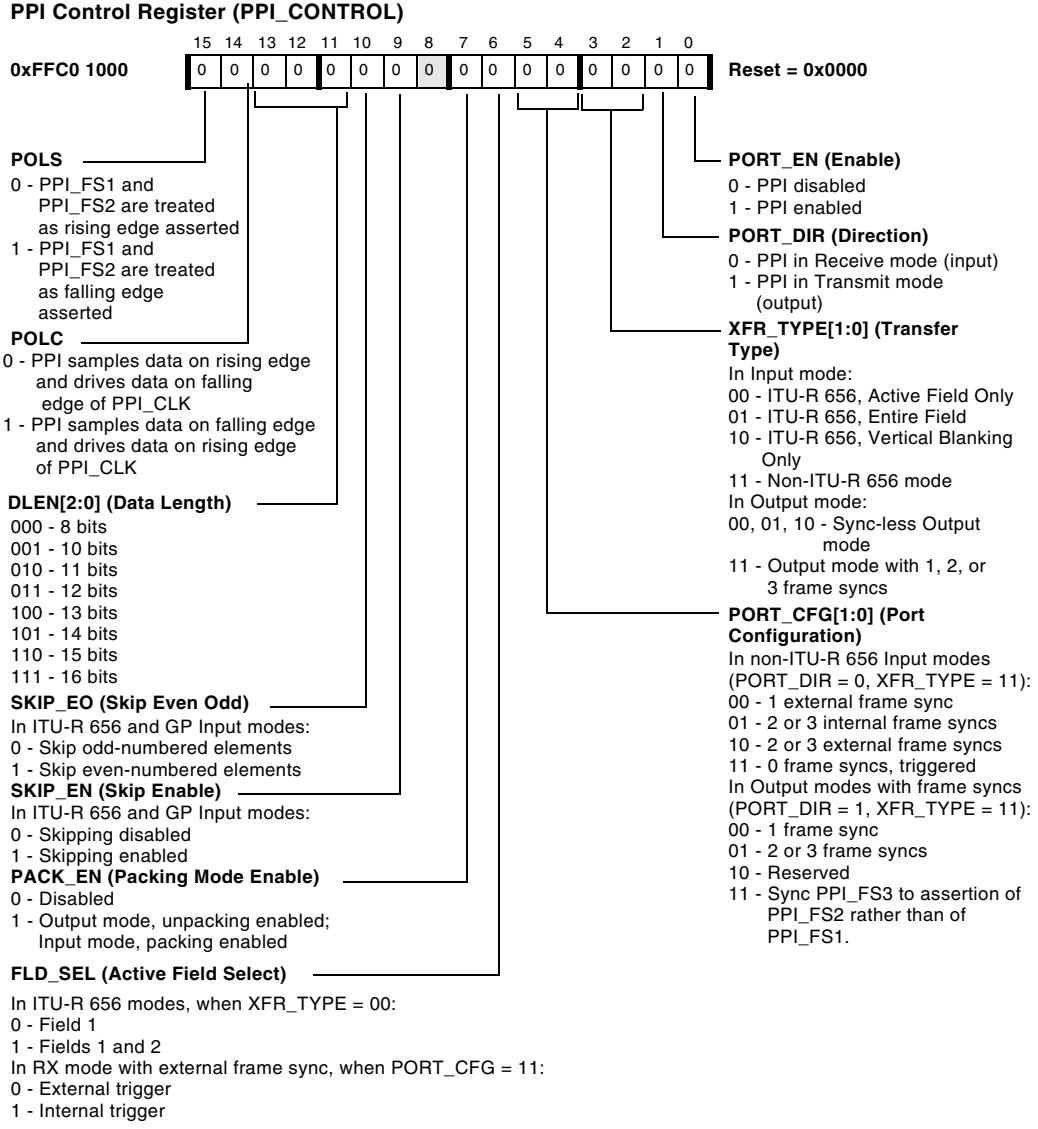


Figure 11-1. PPI Control Register

The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is 8 bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of 10 bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of 8 bits also results in a 16-bit input word, with the upper 8 bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:

0xCE, 0xFA, 0xFE, 0xCA....

- With PACK_EN set:

  This is read into the PPI, configured for an 8-bit port width:
0xCE, 0xFA, 0xFE, 0xCA...

- This is transferred onto the DMA bus:
0xFACE, 0xCAFE, ...

- With PACK_EN cleared:

  This is read into the PPI:
0xCE, 0xFA, 0xFE, 0xCA, ...

- This is transferred onto the DMA bus:
0x00CE, 0x00FA, 0x00FE, 0x00CA, ...

For TX modes, setting PACK_EN enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

0xFACE CAFE.... (0xFA and 0xCA are the two Most Significant Bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMAed to the PPI:

```
0xFACE, 0xCAFE, ...
```

- This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

```
0xCE, 0xFA, 0xFE, 0xCA, ...
```

- With PACK_EN cleared:

  This is DMAed to the PPI:

```
0xFACE, 0xCAFE, ...
```

- This is transferred out through the PPI, configured for an 8-bit port width:

```
0xCE, 0xFE, ...
```

The FLD_SEL bit is used primarily in the Active Field Only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only Field 1 of each video frame, or both Fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 11-2 for the possible operating modes for the PPI.

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 11-2 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

Table 11-2. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	X
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	X
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	X
RX mode, ITU-R 656, Active Field Only	embedded	0	XX	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, Vertical Blanking Only	embedded	0	XX	10	0 or 1	0	X
RX mode, ITU-R 656, Entire Field	embedded	0	XX	01	0 or 1	0	X
TX mode, 0 frame syncs	0	1	XX	00, 01, 10	0 or 1	0 or 1	X
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	X
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	X
TX mode, 2 or 3 internal frame syncs, FS3 sync'ed to FS1 assertion	3	1	01	11	0 or 1	0 or 1	X
TX mode, 2 or 3 internal frame syncs, FS3 sync'ed to FS2 assertion	3	1	11	11	0 or 1	0 or 1	X

Note that, when configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (Timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 11-28 for more information.

PPI_STATUS Register

The PPI status register (PPI_STATUS) contains bits that provide information about the current operating state of the PPI.

The entire register is cleared when read, so the status word must be parsed to evaluate which bits have been set.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASKx register.

The FT_ERR bit is sticky and indicates, when set, that a Frame Track Error has occurred. It is valid for RX modes only. In this condition, the programmed number of lines per frame in PPI_FRAME does not match up with the "frame start detect" condition (see the information note on page -11). A Frame Track Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASKx register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for Input modes only. The state of FLD reflects the current state of the F or PPI_FS3 signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO Overflow Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASKx register.

The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO Underrun Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASKx register.

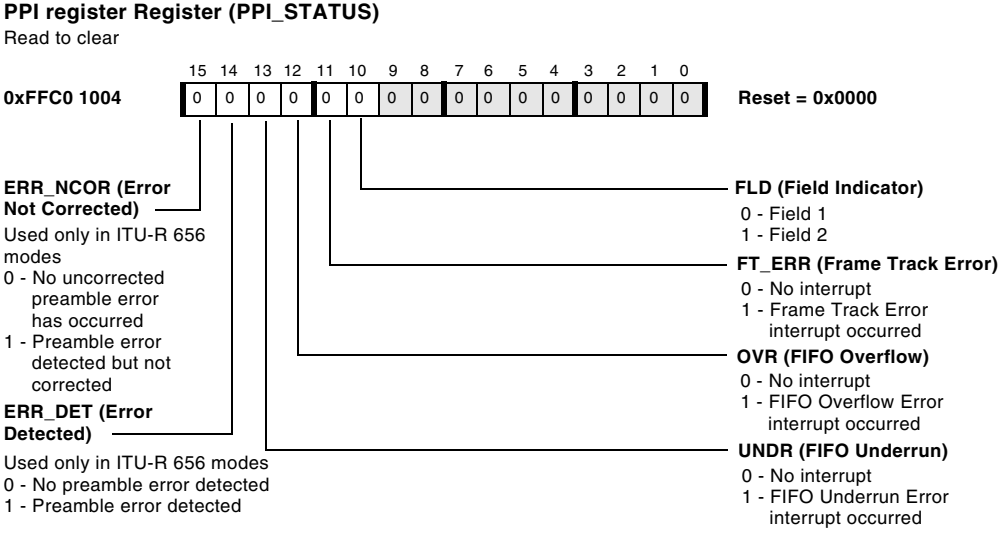


Figure 11-2. PPI Status Register

PPI_DELAY Register

The Delay Count register (PPI_DELAY) can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

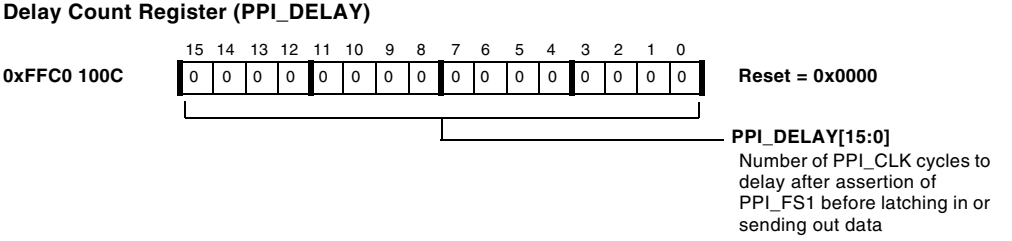


Figure 11-3. Delay Count Register

PPI_COUNT Register

The Transfer Count register (PPI_COUNT) is used only in cases where recurring hardware frame syncs (either externally or internally generated) are involved. It is not needed in ITU-R 656 modes or modes with 0 frame syncs. For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the “horizontal” interval specified by PPI_FS1.

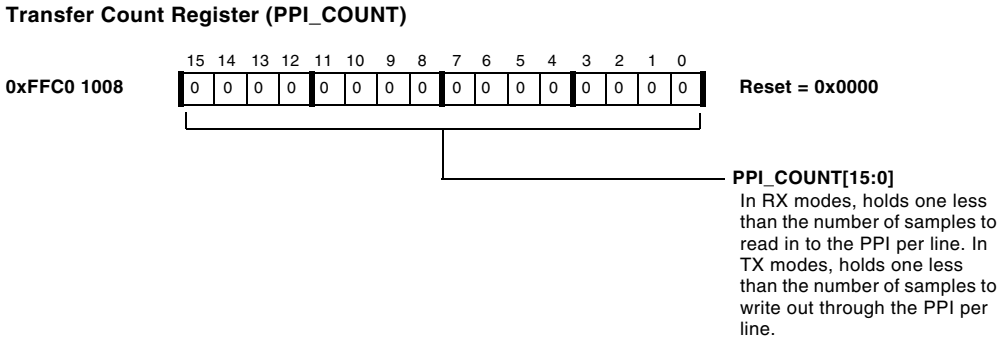


Figure 11-4. Transfer Count Register

PPI_FRAME Register

The Lines Per Frame (PPI_FRAME) register is used in all TX and RX modes with external frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as Field 1 and Field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non-ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original "frame start" each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a Frame Track Error results, and the FT_ERR bit in PPI_STATUS is set. However, the PPI still automatically re-initializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the Field indicator. This occurs at the start of Field 1.

In RX mode with 3 external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with 3 external frame syncs, and only 2 syncs are needed, configure the PPI for three-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

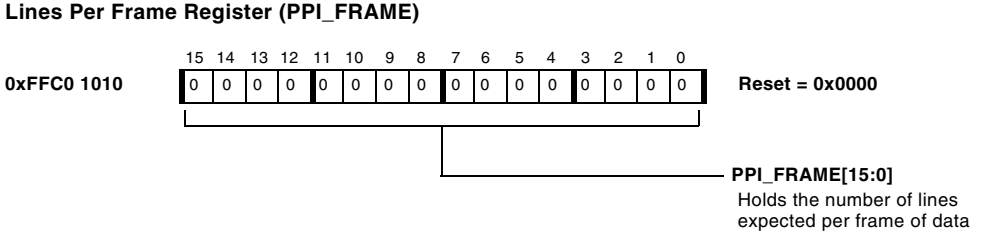


Figure 11-5. Lines Per Frame Register

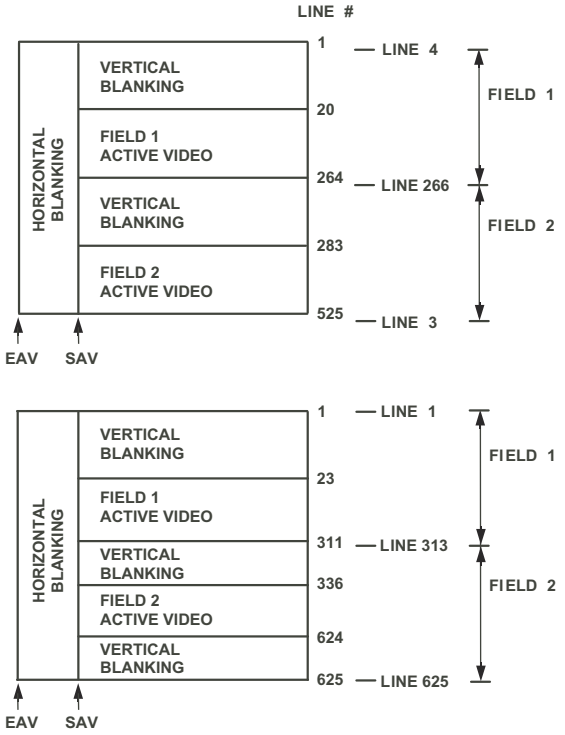
ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 11-6 on page -15, and Figure 11-7 on page -14 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the Bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the Horizontal (H), Vertical (V), and Field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The Start of Active Video (SAV) and End of Active Video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a 0-to-1 transition of H. An entire field of video is comprised of Active Video + Horizontal Blanking (the space between an EAV and SAV code) and Vertical Blanking (the space where V = 1). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between Field 1 and Field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0

LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 11-7. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 11-3. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY register word, which, aside from the `F` (Field), `V` (Vertical Blanking) and `H` (Horizontal Blanking) bits, contains four protection bits for single-bit error detection and correction. Note `F` and `V` are only allowed to change as part of EAV sequences (that is, transition from `H = 0` to `H = 1`). The bit definitions are as follows:

- `F = 0` for Field 1
- `F = 1` for Field 2

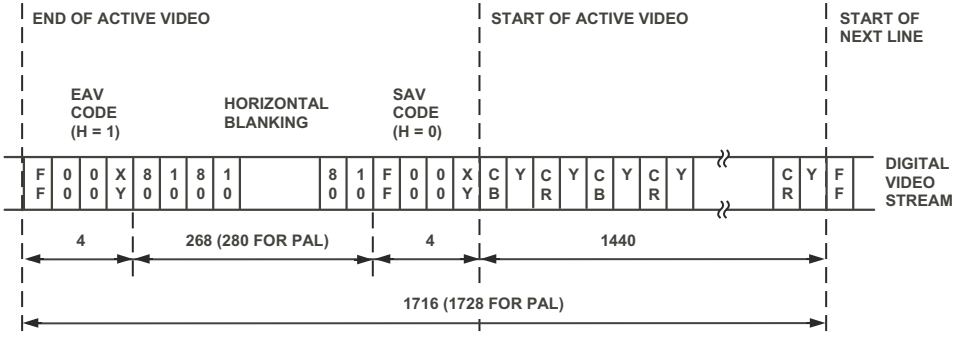


Figure 11-6. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems

- V = 1 during Vertical Blanking
- V = 0 when not in Vertical Blanking
- H = 0 at SAV
- H = 1 at EAV
- P3 = V XOR H
- P2 = F XOR H
- P1 = F XOR V
- P0 = F XOR V XOR H

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the V and F codes can be used to delimit fields and frames.

Table 11-3. Control Byte Sequences for 8-bit and 10-bit ITU-R 656 Video

	8-bit Data								10-bit Data	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 11-8 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.

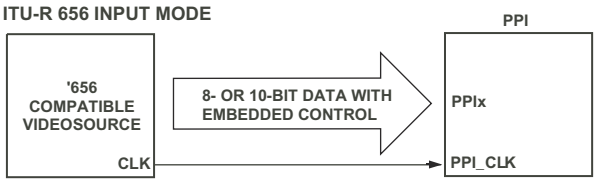


Figure 11-8. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: Entire Field, Active Video Only, and Vertical Blanking Interval Only. Figure 11-9 shows these three submodes.

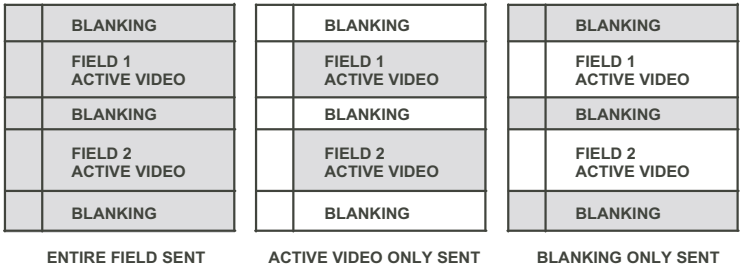


Figure 11-9. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes Active Video as well as control byte sequences and ancillary data that may be embedded in Horizontal and Vertical Blanking Intervals. Data transfer starts immediately after synchronization to Field 1 occurs, but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a “loopback” feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of Field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the PPI_FRAME MMR.

Vertical Blanking Interval (VBI) only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the Vertical Blanking Interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (Active plus Vertical Blanking) per frame in the PPI_FRAME MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of Field 1, which doesn't necessarily correspond to the start of Vertical Blanking. For instance, in 525/60 systems, the start of Field 1 (F = 0) corresponds to Line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. How-

ever, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of Field 1. Consequently, up to two fields might be ignored (for example, if Field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signalling is embedded in the datastream in ITU-R 656 modes, the `PPI_COUNT` register is not necessary. However, the `PPI_FRAME` register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in `PPI_FRAME`. If there is a mismatch, the `FT_ERR` bit in the `PPI_STATUS` register is asserted. For instance, if an SAV transition is missed, the current field will only have `NUM_ROWS - 1` rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the Field register bit is toggled in the `PPI_STATUS` register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The General-Purpose (GP) PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 11-4 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed processor function (that is, as a timer or flag pin). The exception to this is that when the PPI is configured for a 2-frame-sync mode, PPI_FS3 cannot be used as a general-purpose flag, even though it is not used by the PPI.

Table 11-4. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input
RX mode, 2 or 3 external frame syncs	Input	Input	Input	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output
TX mode, 2 external frame syncs	Input	Input	Output	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output	Output

Figure 11-10 illustrates the general flow of the GP modes. The top of the diagram shows an example of RX mode with 1 external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT.

Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter re-initializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 11-10 shows an example of TX mode, 1 internal frame sync. After PPI_FS1 is asserted, there is a latency of 1 PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

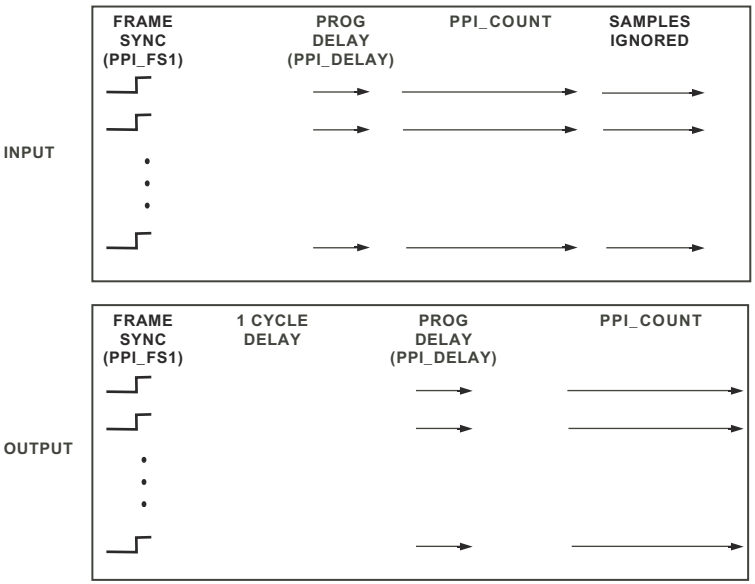


Figure 11-10. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 on page -7 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.
- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least 2 PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 11-11 shows a typical illustration of the system setup for this mode.

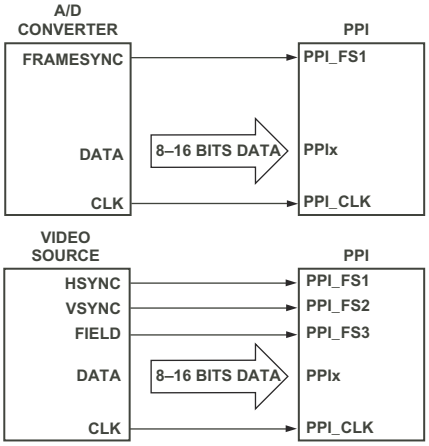


Figure 11-11. RX Mode, External Frame Syncs

The 3-sync mode shown at the bottom of Figure 11-11 supports video applications that use hardware signalling (`HSYNC`, `VSYNC`, `FIELD`) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is `PPI_FS1 = HSYNC`, `PPI_FS2 = VSYNC`, `PPI_FS3 = FIELD`. Refer to "Frame Synchronization in GP Modes" on page 11-28 for more information about frame syncs in this mode.

A 2-sync mode is implicitly supported by pulling `PPI_FS3` to `GND` by an external resistor when configured in 3-sync mode.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 11-12 shows a sample application for this mode.

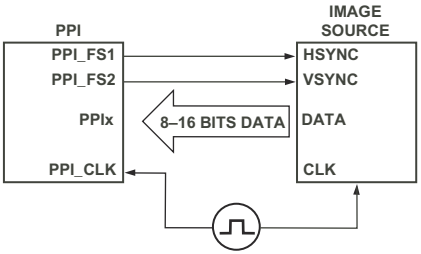


Figure 11-12. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 on page -7 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 11-13 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

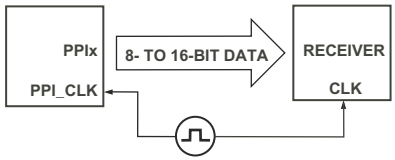


Figure 11-13. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 11-14 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 PPI_CLK cycles, plus the value programmed in PPI_DELAY, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

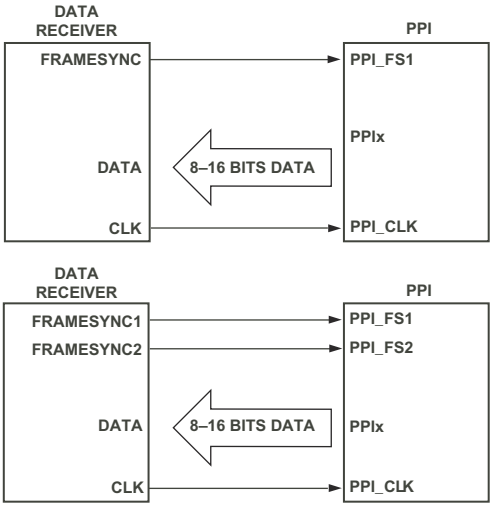


Figure 11-14. TX Mode, 1 or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 11-15 on page -28 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 11-15. A 2-sync mode is implicitly supported by leaving `PPI_FS3` unconnected in this case.

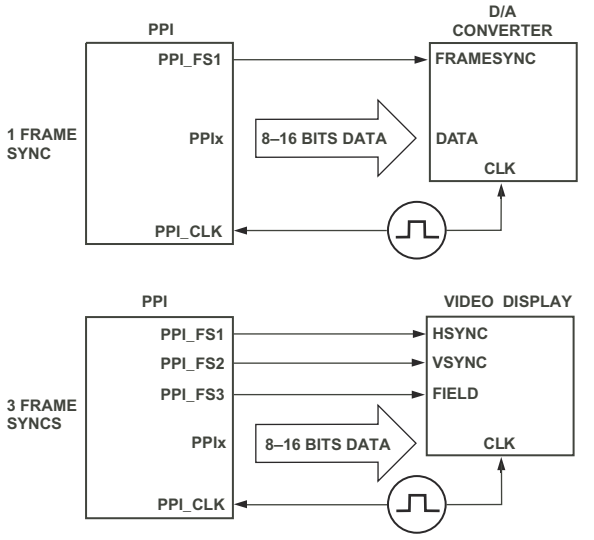


Figure 11-15. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in GP modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes with Internal Frame Syncs

In modes with internal frame syncs, `PPI_FS1` and `PPI_FS2` link directly to the Pulsewidth Modulation (PWM) circuits of Timer 1 and Timer 2, respectively. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing `TIMERx` registers. This capability accommodates a wide range of timing needs.

Note these PWM circuits are clocked by PPI_CLK, not by SCLK or PF1 (as during conventional Timer PWM operation). If PPI_FS2 is not used in the configured PPI mode, Timer 2 operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 11-31.
2. Configure the width and period for each frame sync signal via TIMER1_WIDTH and TIMER1_PERIOD (for PPI_FS1), or TIMER2_WIDTH and TIMER2_PERIOD (for PPI_FS2).
3. Set up TIMER1_CONFIG for PWM_OUT mode (for PPI_FS1). If used, configure TIMER2_CONFIG for PWM_OUT mode (for PPI_FS2). This includes setting CLK_SEL = 1 and TIN_SEL = 1 for each timer.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable Timer 1 and/or Timer 2.

It is important to guarantee proper frame sync polarity between the PPI and Timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in TIMER1_CONFIG and TIMER2_CONFIG. Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in TIMER1_CONFIG and TIMER2_CONFIG.

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the timers (using TIMER_DISABLE).

Modes with External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such a mode, Timers 1 and 2 can be used for a purpose not involving the TMR1 and TMR2 pins. However, timer access to a TMRx pin is disabled when the PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, Timer 2 is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR2 pin becomes available for timer use as well). For more information on configuring and using the timers, refer to Chapter 16, "Timers".

In RX Mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMERx_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see Chapter 16, "Timers".

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, refer to Chapter 9, "Direct Memory Access".

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of `(PPI_CLK) x (16 bits/transfer)`. In modes where data lengths are greater than 8 bits, only one element can be clocked in per `PPI_CLK` cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and `PACK_EN = 1` (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA Error occurs. In fact, the specification of the `DMAx_XCOUNT` and `DMAx_YCOUNT` MMRs allows for flexible data interrupt points. For example, assume the DMA registers `XMODIFY = YMODIFY = 1`. Then, if a data frame contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting `XCOUNT = 320`, `YCOUNT = 240`, and `DI_SEL = 1` (the `DI_SEL` bit is located in `DMAx_CONFIG`) will interrupt on every row transferred, for the entire frame.

- Setting `XCOUNT = 320`, `YCOUNT = 240`, and `DI_SEL = 0` will interrupt only on the completion of the frame (when 240 rows of 320 bytes have been transferred).
- Setting `XCOUNT = 38,400` (320 x 120), `YCOUNT = 2`, and `DI_SEL = 1` will cause an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

Following is the general procedure for setting up DMA operation with the PPI. Refer to "DMA and Memory DMA MMRs" on page 9-3 for details regarding configuration of DMA.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a `1` to bit 0 in `PPI_CONTROL`.

Data Transfer Scenarios

Figure 11-16 shows two possible ways to use the PPI to transfer in video. These diagrams are very generalized, and bandwidth calculations must be made only after factoring in the exact PPI mode and settings (for example, transfer Field 1 only, transfer odd and even elements).

The top part of the diagram shows a situation appropriate for, as an example, JPEG compression. The first N rows of video are DMAed into L1 memory via the PPI. Once in L1, the compression algorithm operates on the data and sends the compressed result out from the processor via the SPORT. Note that no SDRAM access was necessary in this approach.

The bottom part of the diagram takes into account a more formidable compression algorithm, such as MPEG-2 or MPEG-4. Here, the raw video is transferred directly into SDRAM. Independently, a memory DMA channel transfers data blocks between SDRAM and L1 memory for intermediate processing stages. Finally, the compressed video exits the processor via the SPORT.

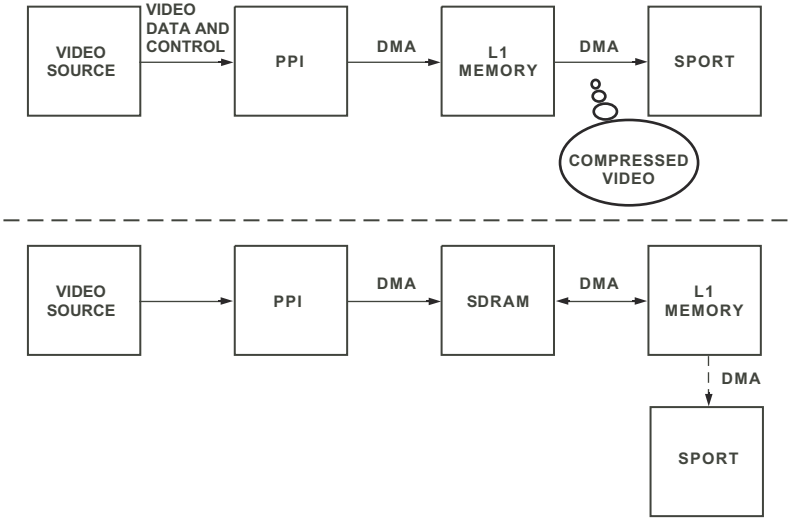


Figure 11-16. PPI Possible Data Transfer Scenarios

12 SERIAL PORT CONTROLLERS

The ADSP-BF539 processor has four identical serial ports (SPORT). These support a variety of serial data communications protocols and can provide a direct interconnection between processors in a multiprocessor system.

The serial ports (SPORT0, SPORT1, SPORT2, SPORT3) provide an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only; the processor provides asynchronous RS-232 data transfer via the UART. Each SPORT has one group of pins (primary data, secondary data, clock, and frame sync) for transmit and a second set of pins for receive. The receive and transmit functions are programmed separately. Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. The SPORTs can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

The naming conventions for registers and pins use a lower case `x` to represent a digit. For example, `RFSx` indicates pins `RFS0`, `RFS1`, `RFS2`, or `RFS3`, corresponding to SPORT0, SPORT1, SPORT2 and SPORT3. LSB refers to least significant bit, and MSB refers to most significant bit.

All SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

The SPORTs use frame sync pulses to indicate the beginning of each word or packet, and the bit clock marks the beginning of each data bit. External bit clock and frame sync are available for the TX and RX buffers.

With a range of clock and frame synchronization options, the SPORTs allow a variety of serial communication protocols, including H.100, and provide a glueless hardware interface to many industry-standard data converters and codecs.

The SPORTs can operate at up to an SCLK/2 clock rate with an externally generated clock, or half the system clock rate for an internally generated serial port clock. The SPORT external clock must always be less than the SCLK frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

SPORT clocks and frame syncs can be internally generated by the system or received from an external source. The SPORTs can operate with a transmission format of LSB first or MSB first, with word lengths selectable from 3 to 32 bits. They offer selectable transmit modes and optional μ-law or A-law companding in hardware. SPORT data can be automatically transferred between on-chip and off-chip memories using DMA block transfers. Additionally, each of the SPORTs offers a TDM (Time-Division-Multiplexed) Multichannel mode.

Each of the SPORTs offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.

- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a Shift register), providing additional time to service the SPORT.
- Provides two synchronous transmit and two synchronous receive data pins and buffers in each SPORT to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 12-35 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).
- Executes DMA transfers to and from on-chip memory. Each SPORT can automatically receive and transmit an entire block of data.

- Permits chaining of DMA operations for multiple data blocks.
- Has a multichannel mode for TDM interfaces. Each SPORT can receive and transmit data selectively from a Time-Division-Multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 = (1023 – 128). Note the Multichannel Select registers and the `WSIZE` register control which subset of the 128 channels within the active region can be accessed.

Table 12-1 shows the pins for each SPORT.

Table 12-1. Serial Port (SPORT) Pins

Pin[1]	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

1 A lowercase x within a pin name represents a possible value of 0, 1, 2, or 3 (corresponding to SPORT0, SPORT1, SPORT2, or SPORT3).

SPORT0 and SPORT1 have dedicated pins, whereas SPORT2 and SPORT3 utilize GPIO pins. See Chapter 15, “General Purpose Input/Output Ports”.

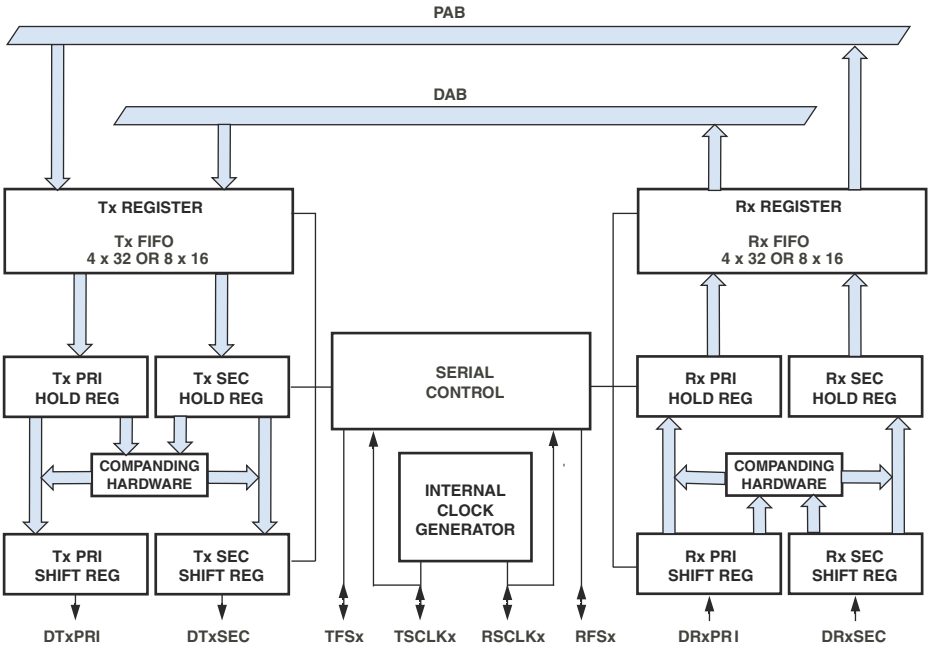
A SPORT receives serial data on its DRxPRI and DRxSEC inputs and transmits serial data on its DTxPRI and DTxSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTxPRI and DTxSEC) are synchronous to the transmit clock (TSCLKx). For receive, the data bits (DRxPRI and DRxSEC) are synchronous to the receive clock (RSCLKx). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFSx and TFSx are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary pins is interleaved in main memory and can be retrieved by setting a stride in the Data Address Generators (DAG) unit. For more information about DAGs, see Chapter 5, “Data Address Generators”. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor’s powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

The following figure shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT’s SPORTx_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX Shift register. The bits in the Shift register are shifted out on the SPORT’s DTxPRI/DTxSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLKx pin. The receive portion of the SPORT accepts data from the DRxPRI/DRxSEC pin synchronous to the serial clock on the RSCLKx pin.

When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's SPORTx_RX register, and then into the RX FIFO where it is available to the processor.



NOTE 1: ALL WIDE ARROW DATA PATHS ARE 16 OR 32 BITS WIDE, DEPENDING ON SLEN. FOR SLEN = 2 TO 15, A 16-BIT DATA PATH WITH 8-DEEP FIFO IS USED. FOR SLEN = 16 TO 31, A 32-BIT DATA PATH WITH 4-DEEP FIFO IS USED.
NOTE 2: Tx REGISTER IS THE BOTTOM OF THE Tx FIFO, Rx REGISTER IS THE TOP OF THE Rx FIFO.

Figure 12-1. SPORT Block Diagram

Figure 12-2 shows a possible port connection for the SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices 1, 2, and N.

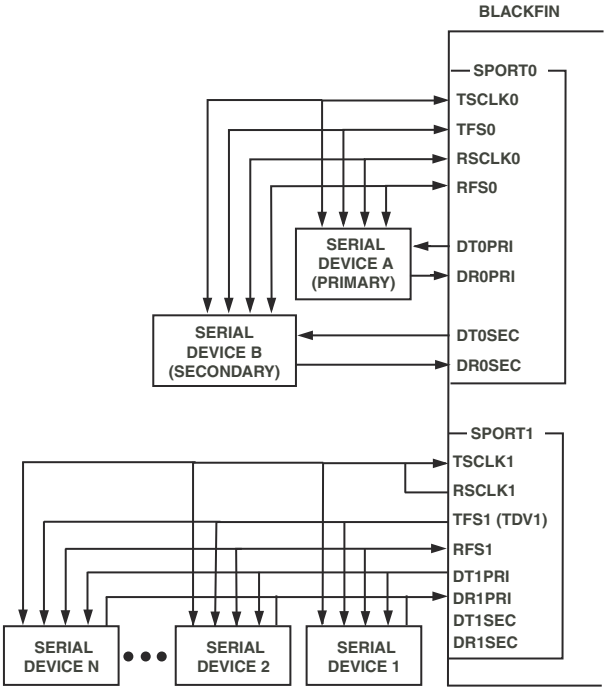


Figure 12-2. SPORT Connections[1, 2]

1 In multichannel mode, TFS1 functions as a transmit data valid (TDV1) output. See "Multichannel Operation" on page 12-49 for details.

2 Although shown as an external connector, the TSCLK1/RSCLK1 connection is internal in multichannel mode. See "Multichannel Operation" on page 12-49 for details.

Figure 12-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

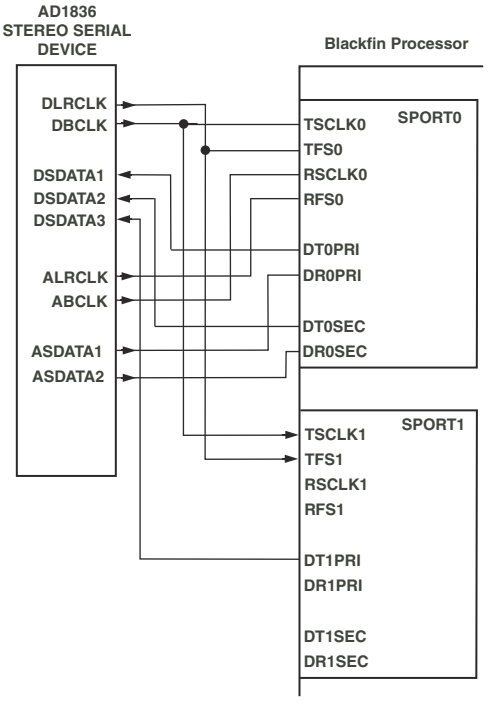


Figure 12-3. Stereo Serial Connection

SPORT Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT's SPORTx_TX register readies the SPORT for transmission. The TFSx signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORTx_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORTx_TCR1 register. Each bit is shifted out on the driving edge of TSCLKx. The driving edge of TSCLKx can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed. For information about DMA, see Chapter 9, "Direct Memory Access".

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORTx_TCR1 register and RSPEN in the SPORTx_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORTx_TCLKDIV, SPORTx_RCLKDIV, SPORTx_TFSDIV, and SPORTx_RFSDIV clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFSx/RFSx and TSCLKx/RSCLKx pins, if these signals are configured to be generated internally.

When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. Each SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT Configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TCR1 and/or RSPEN in SPORTx_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the following sections. All control and status bits in the SPORT registers are active high unless otherwise noted.

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). The SPORTx_TX register writes are always enabled; SPORTx_RX, SPORTx_CHNL, and SPORTx_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORTx_TCR1 and SPORTx_TCR2.

A SPORT is enabled for transmit if Bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORTx_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_TCR1 is not written except for bit 0 (TSPEN). For example:

```
write (SPORTx_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORTx_TCR1, 0xFF01) ;   /* ignored, no effect */
write (SPORTx_TCR1,0xFFF0);   /*SPORTdisabled,SPORTx_TCR1
still equal to 0x0000 */
```

The addresses for these SPORT registers are:

SPORT0_TCR1 – 0xFFC0 0800	SPORT0_TCR2 – 0xFFC0 0804
SPORT1_TCR1 – 0xFFC0 0900	SPORT1_TCR2 – 0xFFC0 0904
SPORT2_TCR1 – 0xFFC0 2500	SPORT2_TCR2 – 0xFFC0 2504
SPORT3_TCR1 – 0xFFC0 2600	SPORT3_TCR2 – 0xFFC0 2604

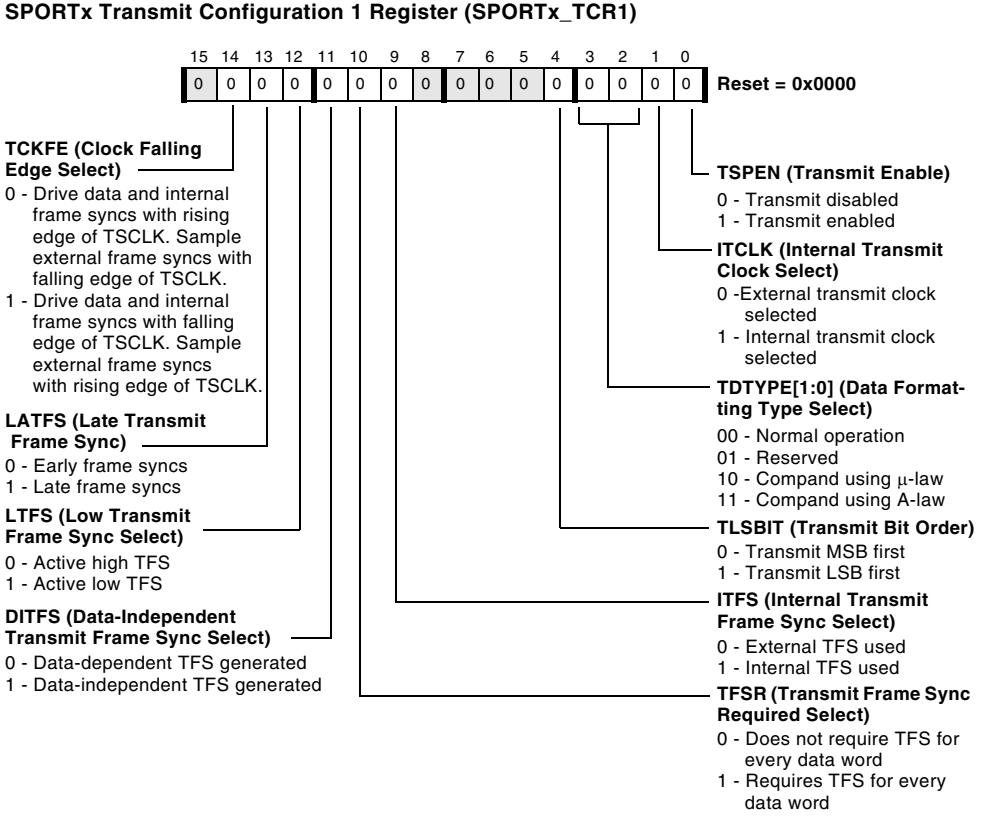


Figure 12-4. SPORTx Transmit Configuration 1 Register

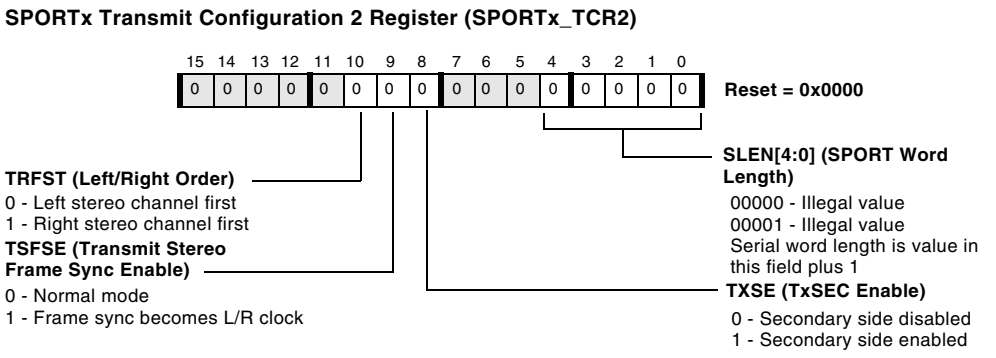


Figure 12-5. SPORTx Transmit Configuration 2 Register

Additional information for the SPORTx_TCR1 and SPORTx_TCR2 Transmit Configuration register bits includes:

- **Transmit Enable** (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting TSPEN causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting TSPEN.

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting TSPEN. Set all DMA control registers before setting TSPEN.

  Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_TCR1 with all of the necessary bits, including TSPEN.

- **Internal Transmit Clock Select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data Formatting Type Select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit Order Select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial Word Length Select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

```
Serial Word Length = SLEN + 1;
```

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The serial port transfers bits [SLEN:0] from the transmit buffer.

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal Transmit Frame Sync Select**. (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit Frame Sync Required Select**. (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a Transmit Frame Sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent Transmit Frame Sync Select**. (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORTx_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORTx_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORTx_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the SPORTx_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORTx_TX register.

- **Low Transmit Frame Sync Select**. (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).
- **Late Transmit Frame Sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock Drive/Sample Edge Select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec Enable**. (TXSE). This bit enables the transmit secondary side of the serial port (if set).
- **Stereo Serial Enable**. (TSFSE). This bit enables the stereo serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right Order**. (TRFST). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORTx_RCR1 and SPORTx_RCR2.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT Configuration register writes are not allowed except for SPORTx_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_RCR1 is not written except for bit 0 (RSPEN). For example:

```
write (SPORTx_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORTx_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORTx_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORTx_RCR1
still equal to 0x0000 */
```

The addresses for these SPORT registers are:

SPORT0_RCR1 – 0xFFC0 0820	SPORT0_RCR2 – 0xFFC0 0824
SPORT1_RCR1 – 0xFFC0 0920	SPORT1_RCR2 – 0xFFC0 0924
SPORT2_RCR1 – 0xFFC0 2520	SPORT2_RCR2 – 0xFFC0 2524
SPORT3_RCR1 – 0xFFC0 2620	SPORT3_RCR2 – 0xFFC0 2624

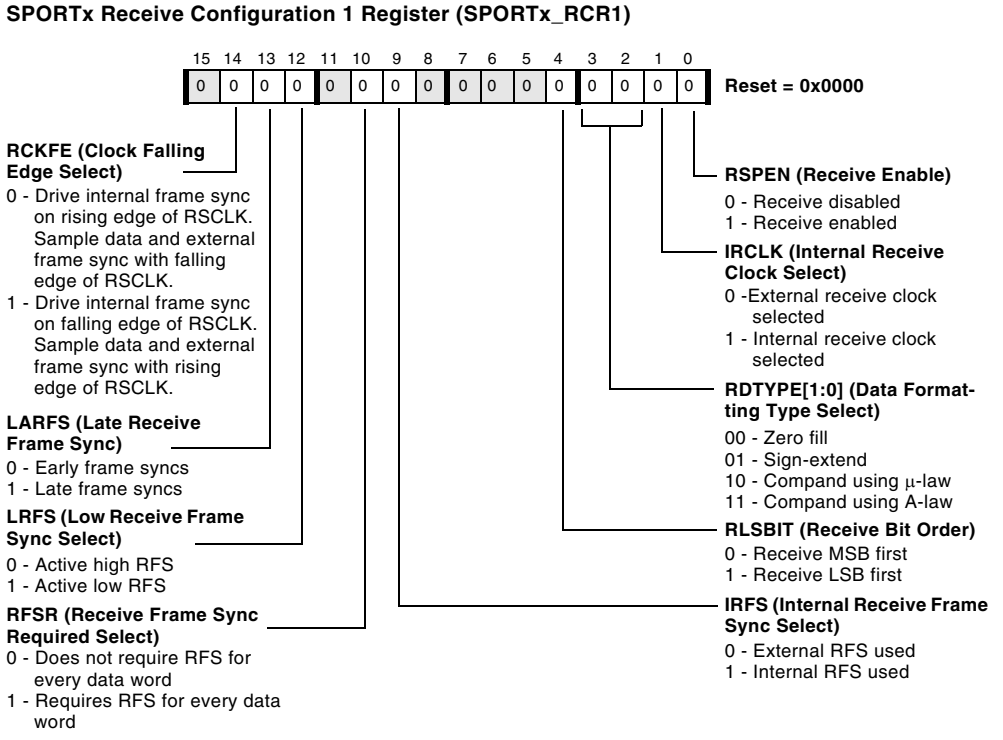


Figure 12-6. SPORTx Receive Configuration 1 Register

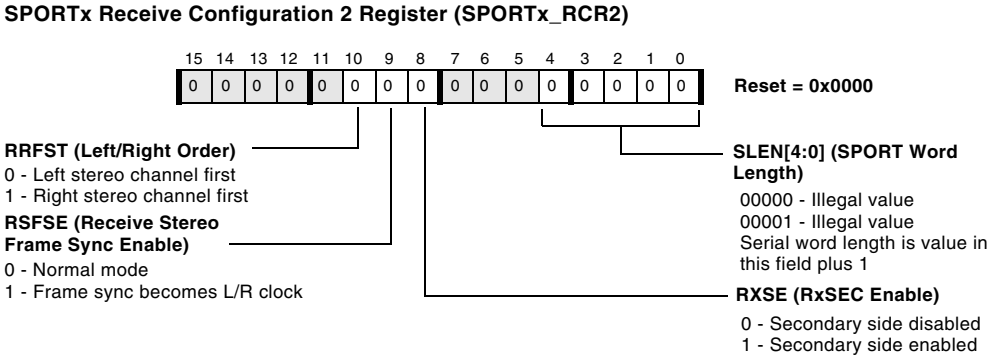


Figure 12-7. SPORTx Receive Configuration 2 Register

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 RFS (if set) or an external RFS (if cleared).
- **Receive Frame Sync Required Select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a Receive Frame Sync for every data word.

- **Low Receive Frame Sync Select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late Receive Frame Sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock Drive/Sample Edge Select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec Enable**. (RXSE). This bit enables the receive secondary side of the serial port (if set).
- **Stereo Serial Enable**. (RSFSE). This bit enables the stereo serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right Order**. (RRFST). If this bit is set, the right channel is received first in Stereo Serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers.

SPORT Transmit Data (SPORTx_TX) Register

The SPORTx transmit data register (SPORTx_TX) is a write-only register. Reads produce a peripheral access bus (PAB) error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length <= 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 12-8:

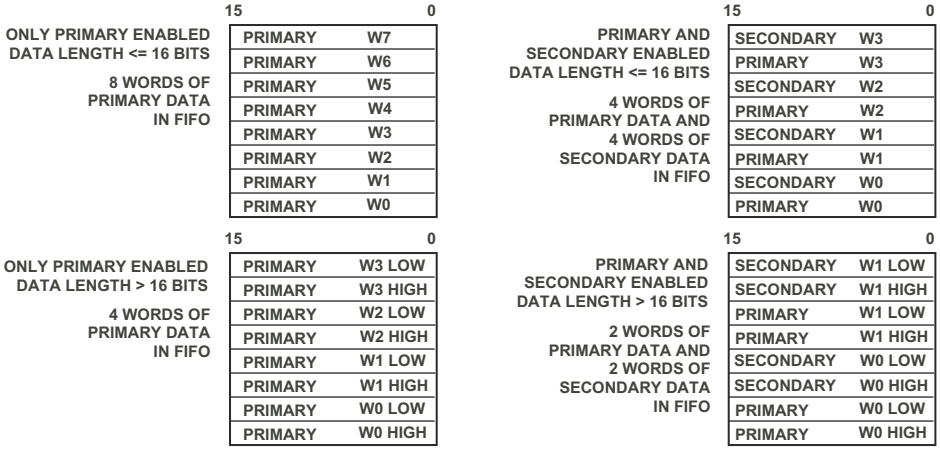


Figure 12-8. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that PAB/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/PAB writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX hold register based on TXSE and SLEN, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the DTxPRI and DTxSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled. For DMA operation, see Chapter 9, "Direct Memory Access".

The transmit underflow register bit (TUVF) is set in the SPORT status register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TSPEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORTx_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORTx_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT status register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORTx status register are updated upon writes from the core processor, even when the SPORT is disabled.

The addresses for these SPORT registers are:

SPORT0_TX – 0xFFC0 0810

SPORT1_TX – 0xFFC0 0910

SPORT2_TX – 0xFFC0 2510

SPORT3_TX – 0xFFC0 2610

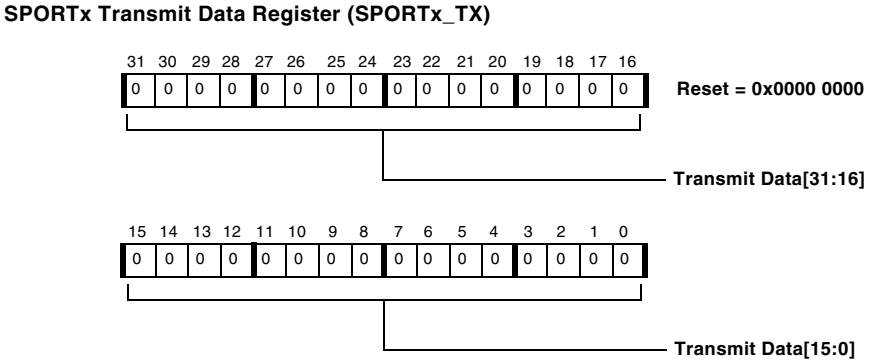


Figure 12-9. SPORTx Transmit Data Register

SPORT Receive Data (SPORTx_RX) Register

The SPORTx receive data register (SPORTx_RX) is a read-only register. Writes produce a PAB error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length <= 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using PAB/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO is shown in Figure 12-10:

The addresses for these SPORT registers are:

SPORT0_RX – 0xFFC0 0818

SPORT1_RX – 0xFFC0 0918

SPORT2_RX – 0xFFC0 2518

SPORT3_RX – 0xFFC0 2618

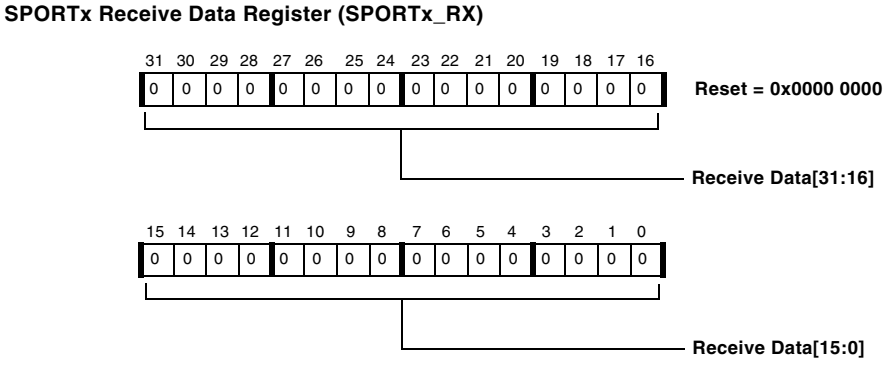


Figure 12-10. SPORTx Receive Data Register

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/PAB reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRxPRI pin is loaded into the RX primary shift register, while data from the DRxSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORTx_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORTx status register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORTx_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT receiver.

SPORT Status (SPORTx_STAT) Register

The SPORT status register (SPORTx_STAT) is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status.

The TXF bit in the SPORT status register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.

The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever

SPORTx_TX is empty by clearing the DITFS control bit in the SPORT configuration register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TSPEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TSPEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT status register. It is a sticky W1C bit and is also cleared by disabling the serial port (writing RSPEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RSPEN = 0. The RUVF bit is updated even when the SPORT is disabled.

The addresses for these SPORT registers are:

SPORT0_STAT – 0xFFC0 0830

SPORT1_STAT – 0xFFC0 0930

SPORT2_STAT – 0xFFC0 2530

SPORT3_STAT – 0xFFC0 2630

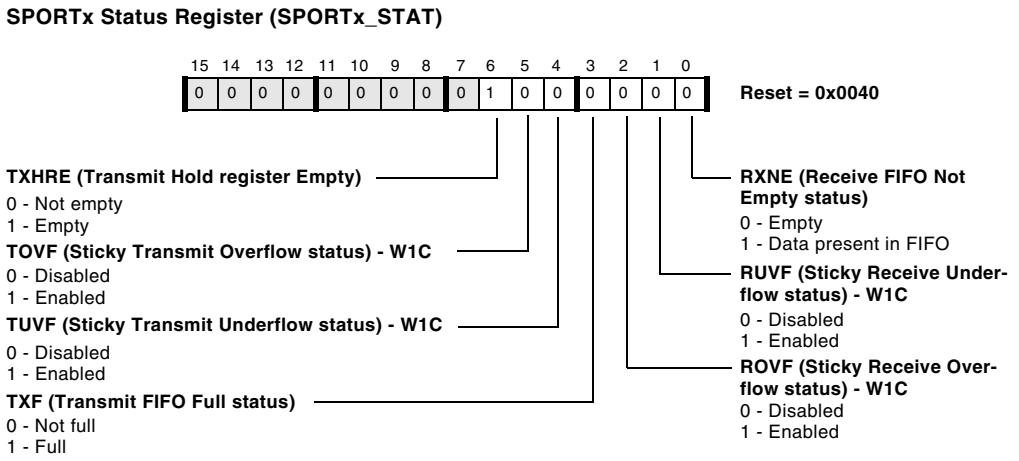


Figure 12-11. SPORTx Status Register

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

PAB Errors

The SPORT generates a PAB error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORTx_TX)
- Writing a read-only register (for example, SPORTx_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORTx transmit serial clock divider register, SPORTx_TCLKDIV, and the SPORTx receive serial clock divider register, SPORTx_RCLKDIV).

The addresses for these SPORT registers are:

SPORT0_TCLKDIV – 0xFFC0 0808	SPORT0_RCLKDIV – 0xFFC0 0828
SPORT1_TCLKDIV – 0xFFC0 0908	SPORT1_RCLKDIV – 0xFFC0 0928
SPORT2_TCLKDIV – 0xFFC0 2508	SPORT2_RCLKDIV – 0xFFC0 2528
SPORT3_TCLKDIV – 0xFFC0 2608	SPORT3_RCLKDIV – 0xFFC0 2628

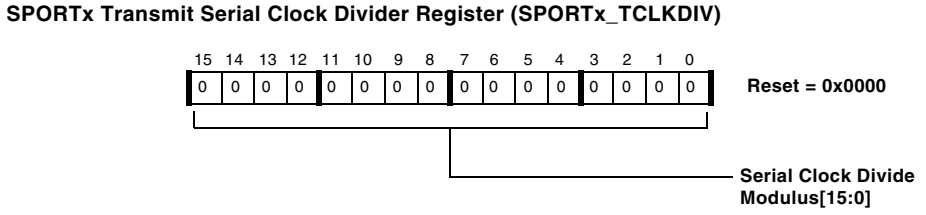


Figure 12-12. SPORTx Transmit Serial Clock Divider Register

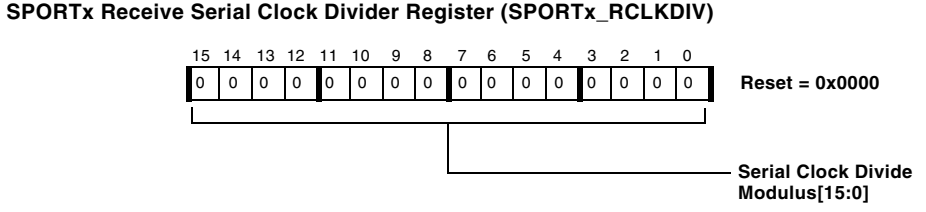


Figure 12-13. SPORTx Receive Serial Clock Divider Register

SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, 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.

The addresses for these SPORT registers are:

SPORT0_TFSDIV – 0xFFC0 080C	SPORT0_RFSDIV – 0xFFC0 082C
SPORT1_TFSDIV – 0xFFC0 090C	SPORT1_RFSDIV – 0xFFC0 092C
SPORT2_TFSDIV – 0xFFC0 250C	SPORT2_RFSDIV – 0xFFC0 252C
SPORT3_TFSDIV – 0xFFC0 260C	SPORT3_RFSDIV – 0xFFC0 262C

SPORTx Transmit Frame Sync Divider Register (SPORTx_TFSDIV)

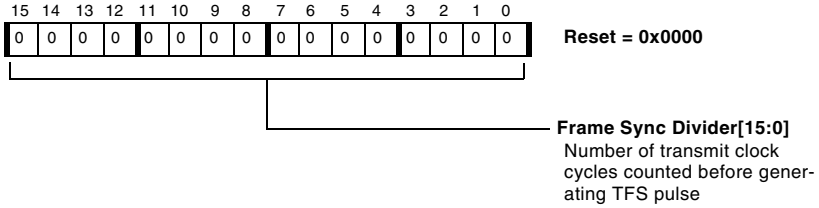


Figure 12-14. SPORTx Transmit Frame Sync Divider Register

SPORTx Receive Frame Sync Divider Register (SPORTx_RFSDIV)

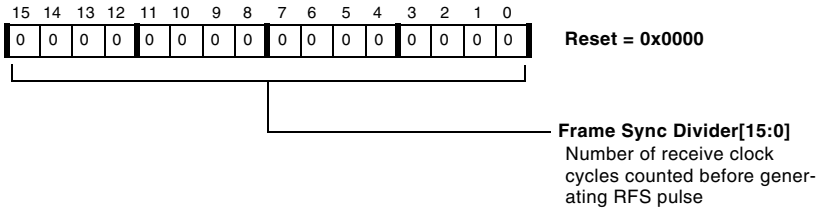


Figure 12-15. SPORTx Receive Frame Sync Divider Register

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORTx_TCLKDIV and SPORTx_RCLKDIV.

```
SPORTx_TCLK frequency =
(SCLK frequency)/(2 x (SPORTx_TCLKDIV + 1))

SPORTx_RCLK frequency =
(SCLK frequency)/(2 x (SPORTx_RCLKDIV + 1))
```

If the value of SPORTx_TCLKDIV or SPORTx_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLKx or RSCLKx that follows the next leading edge of TFSx or RFSx.

When an internal frame sync is selected (ITFS = 1 in the SPORTx_TCR1 register or IRFS = 1 in the SPORTx_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORTx_TCLKDIV or SPORTx_RCLKDIV has changed. The second frame sync will cause the update.

The SPORTx_TFSDIV and SPORTx_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFSx or RFSx pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

```
# of transmit serial clocks between frame sync assertions =
TFSDIV + 1
# of receive serial clocks between frame sync assertions =
RFSDIV + 1
```

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

```
SPORTxTFS frequency = (TSCLKx frequency)/(SPORTx_TFSDIV + 1)

SPORTxRFS frequency = (RSCLKx frequency)/(SPORTx_RFSDIV + 1)
```

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORTx_TCR2 or SPORTx_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late Transmit Frame Syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for exact timing specifications.

Frame Sync & Clock Example

The following code fragment is a brief example of setting up the clocks and frame sync.

```
r0 = 0x00FF;
p0.l = lo(SPORT0_RFSDIV);
p0.h = hi(SPORT0_RFSDIV);
w[p0] = r0.l; ssync;
p0.l = lo(SPORT0_TFSDIV);
w[p0] = r0.l; ssync;
```

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORTx_TCR2 and SPORTx_RCR2 registers of each SPORT determines the word length according to this formula:

```
Serial Word Length = SLEN + 1
```

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN ≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORTx_RCR1 and SPORTx_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORTx_TCR1 register and the RDTYPE field of the SPORTx_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 12-2.

Table 12-2. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORTx_TCR1 Data Formatting	SPORTx_RCR1 Data Formatting
b#00	Normal operation	Zero fill
b#01	Reserved	Sign extend
b#10	Compand using μ-law	Compand using μ-law
b#11	Compand using A-law	Compand using A-law

These formats are applied to serial data words loaded into the SPORTx_RX and SPORTx_TX buffers. SPORTx_TX data words are not actually zero filled or sign extended, because only the significant bits are transmitted.

Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORTs support the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORTx_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORTx_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register.

Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLKx) and a receive clock signal (RSCLKx). The clock signals are configured by the TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers. Serial clock frequency is configured in the SPORTx_TCLKDIV and SPORTx_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORTx_TCR1 configuration register and the IRCLK bit in the SPORTx_RCR1 Configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the core, and the TSCLKx or RSCLKx pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLKx or RSCLKx pins, and the serial clock divisors in the SPORTx_TCLKDIV/SPORTx_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the core system clock or with each other. The core system clock must have a higher frequency than RSCLKx and TSCLKx.

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 the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the processor data sheet for details.

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 12-16 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORTx_TCR1 and SPORTx_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers.

See "Timing Examples" on page 12-68 for more timing examples.

Internal Versus External Frame Syncs

Both Transmit and Receive Frame Syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers determine the frame sync source.

When ITFS = 1 or IRFS = 1, the corresponding frame sync signal is generated internally by the SPORT, and the 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.

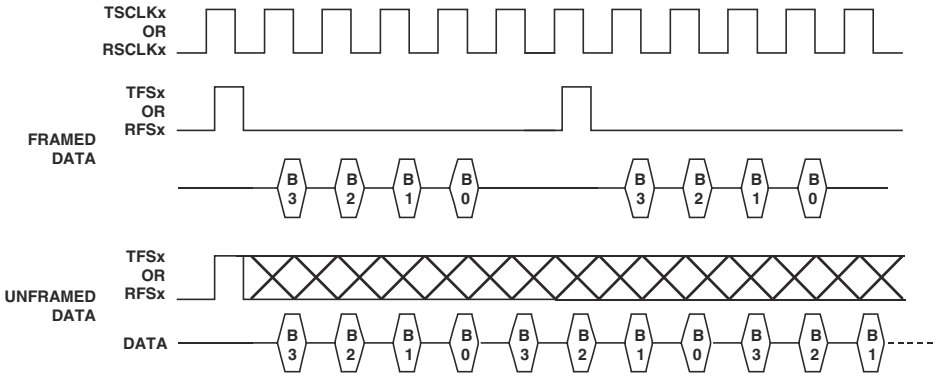


Figure 12-16. Framed Versus Unframed Data

When `ITFS = 0` or `IRFS = 0`, the corresponding frame sync signal is accepted as an input on the `TFSx` pin or `RFSx` pin, and the frame sync divisors in the `SPORTx_TFSDIV`/`SPORTx_RFSDIV` registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The `LTFS` and `LRFS` bits of the `SPORTx_TCR1` and `SPORTx_RCR1` registers determine frame sync logic levels:

- When `LTFS = 0` or `LRFS = 0`, the corresponding frame sync signal is active high.
- When `LTFS = 1` or `LRFS = 1`, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The `LTFS` and `LRFS` bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORTx_TCR1 register selects the falling edge of TSCLKx to drive data and internally generated frame syncs and selects the rising edge of TSCLKx to sample externally generated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLKx to drive data and internally generated frame syncs and selects the falling edge of TSCLKx to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORTx_RCR1 register selects the falling edge of RSCLKx to drive internally generated frame syncs and selects the rising edge of RSCLKx to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLKx to drive internally generated frame syncs and selects the falling edge of RSCLKx to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORTx_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 12-17, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

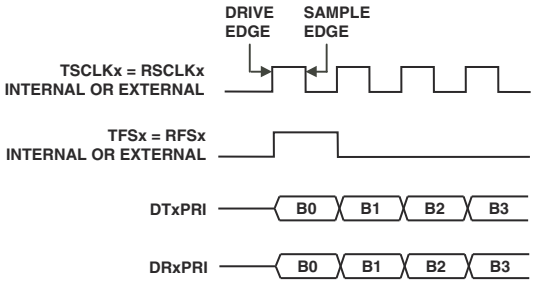


Figure 12-17. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 12-18, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

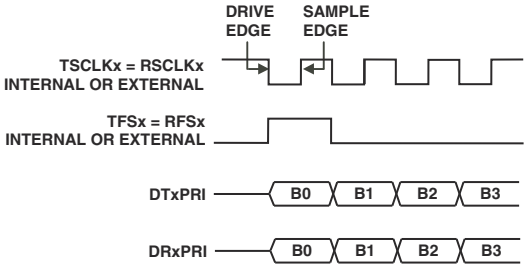


Figure 12-18. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN ≥ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 12-19 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORTx_TCR1 or SPORTx_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See "Timing Examples" on page 12-68 for more examples.

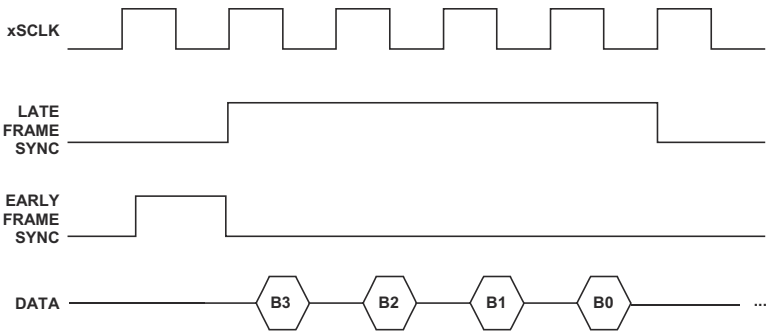


Figure 12-19. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFSx) is output only when the SPORTx_TX buffer has data ready to transmit. The data-independent transmit frame sync select bit (DITFS) allows the continuous generation of the TFSx signal, with or without new data. The DITFS bit of the SPORTx_TCR1 register configures this option.

When DITFS = 0, the internally generated TFSx is only output when a new data word has been loaded into the SPORTx_TX buffer. The next TFSx is generated once data is loaded into SPORTx_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFSx is output at its programmed interval regardless of whether new data is available in the SPORTx_TX buffer. Whatever data is present in SPORTx_TX is transmitted again with each assertion of TFSx. The TUVF (transmit underflow status) bit in the SPORTx_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORTx_TX buffer does not have new data when an externally generated TFSx occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFSx is used, a single write to the SPORTx_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see Chapter 9, "Direct Memory Access".

Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I^2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or SPORT_TCR2 changes the operation of the frame sync pin to a left/right clock as required for I^2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 12-20 on page -46 and Figure 12-21 on page -47 show timing diagrams for stereo serial mode operation.

Table 12-3 shows several modes that can be configured using bits in SPORTx_TCR1 and SPORTx_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

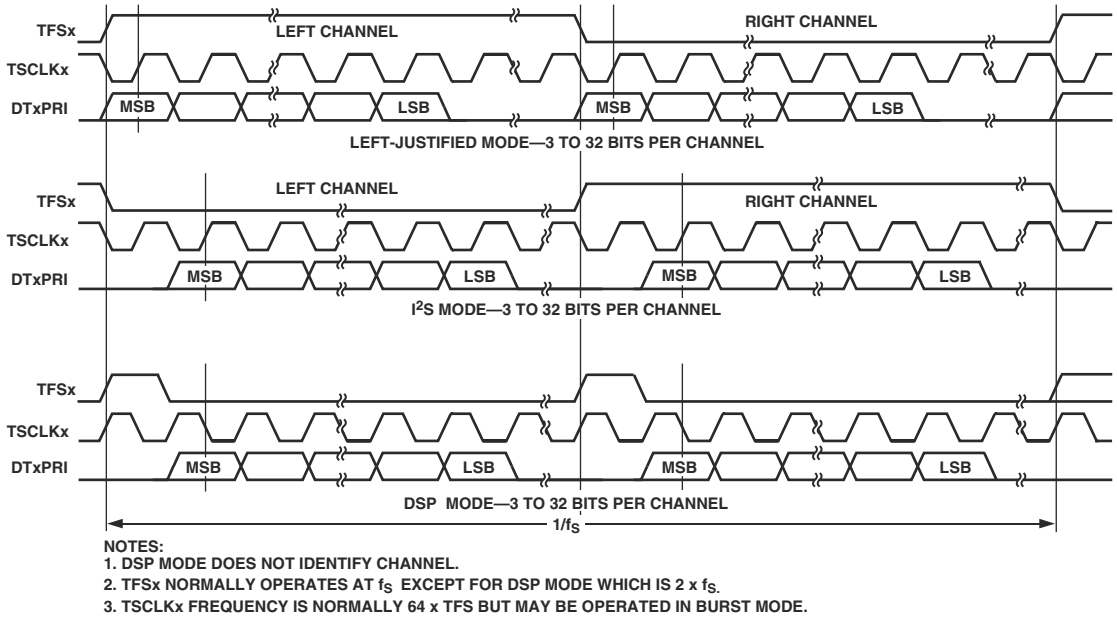


Figure 12-20. SPORT Stereo Serial Modes, Transmit

Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 12-3 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I^{2}S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 12-32) still apply,

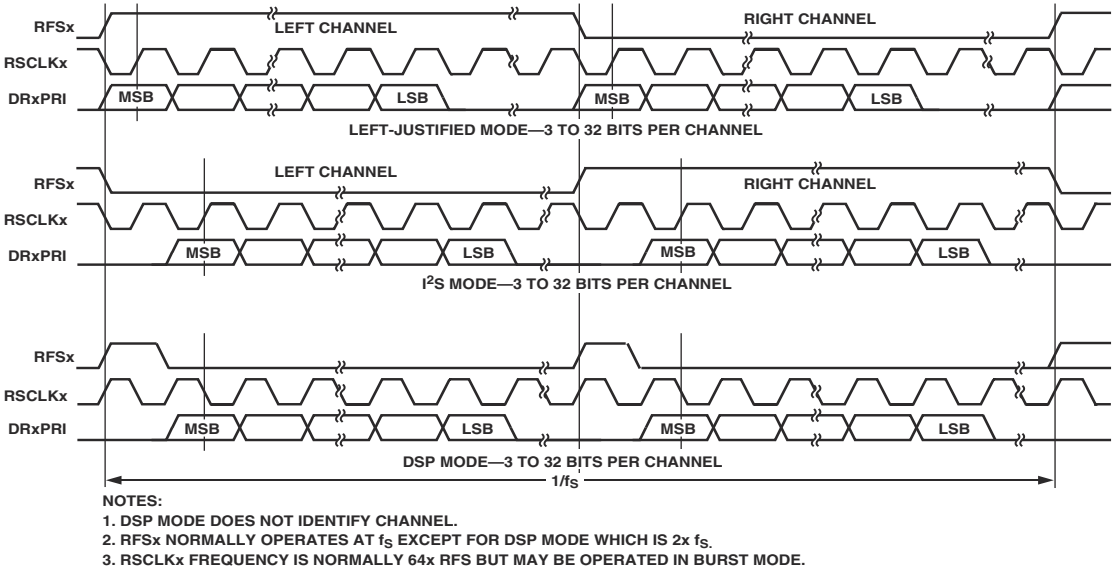


Figure 12-21. SPORT Stereo Serial Modes, Receive

Table 12-3. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I²S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1
RCKFE	1	0	0
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0

Table 12-3. Stereo Serial Settings (Cont'd)

Bit Field	Stereo Audio Serial Scheme		
	I^2S	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 on the RFSx or TFSx pin is the left channel. This is the default setting.

The RRFST and TRFST bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRxSEC and DTxSEC pins are useful extensions of the serial port which pair well with Stereo Serial mode. Multiple I^2S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (Frame Sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 12-3 on page -8, 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 bitstream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024 total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The `DTxPRI` pin is always driven (not three-stated) if the SPORT is enabled (`TSPEN = 1` in the `SPORTx_TCR1` register), unless it is in multichannel mode and an inactive time slot occurs. The `DTxSEC` pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (`TXSE = 1` in the `SPORTx_TCR2` register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

In multichannel mode, RSCLKx can either be provided externally or generated internally by the SPORT, and it is used for both transmit and receive functions. Leave TSCLKx disconnected if the SPORT is used only in multichannel mode. If RSCLKx is externally or internally provided, it will be internally distributed to both the receiver and transmitter circuitry.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORTx_TX or SPORTx_RX operation for Multichannel Mode. This is especially important in DMA data unpacked mode, since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORTx_MCMC2) must be enabled prior to enabling SPORTx_TX or SPORTx_RX operation. When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 12-22 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 12-68 for more examples.

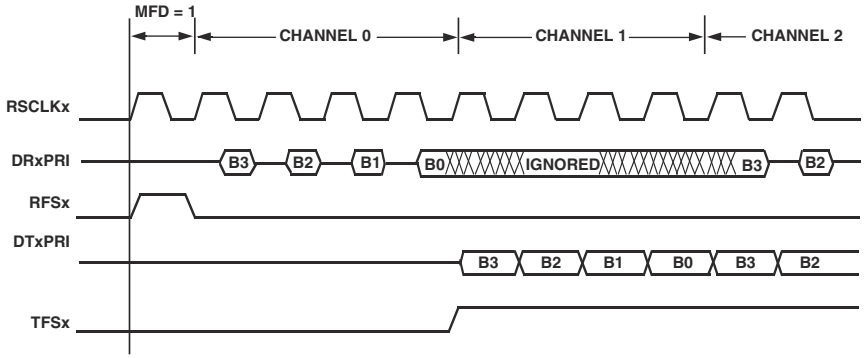


Figure 12-22. Multichannel Operation

SPORT Multichannel Configuration (SPORTx_MCMCn) Registers

There are two SPORTx multichannel configuration registers (`SPORTx_MCMCn`) for each SPORT. The `SPORTx_MCMCn` registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

The addresses for these SPORT registers are:

SPORT0_MCMC1 – 0xFFC0 0838
SPORT0_MCMC2 – 0xFFC0 083C

SPORT1_MCMC1 – 0xFFC0 0938
SPORT1_MCMC2 – 0xFFC0 093C

SPORT2_MCMC1 – 0xFFC0 2538
SPORT2_MCMC2 – 0xFFC0 253C

SPORT3_MCMC1 – 0xFFC0 2638
SPORT3_MCMC2 – 0xFFC0 263C

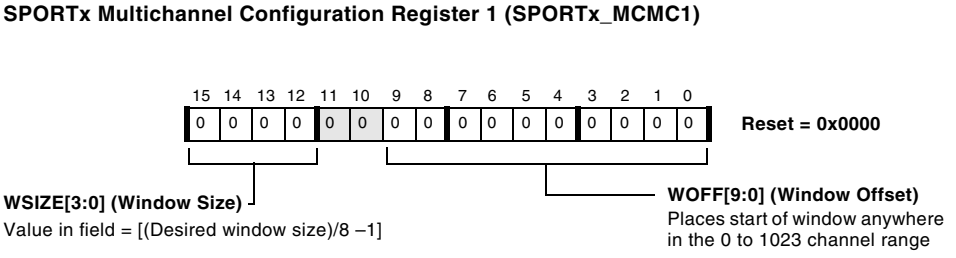


Figure 12-23. SPORTx Multichannel Configuration Register 1

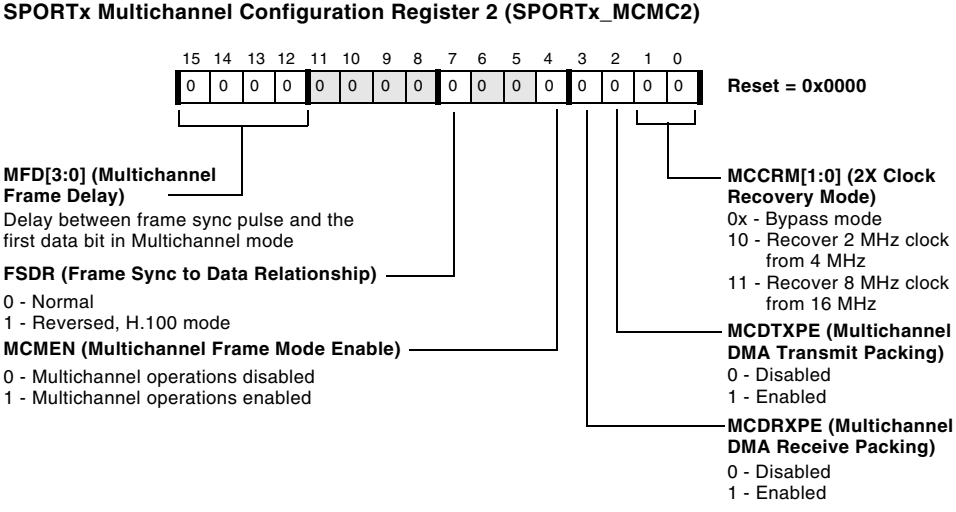


Figure 12-24. SPORTx Multichannel Configuration Register 2

Multichannel Enable

Setting the MCMEN bit in the SPORTx_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 12-4 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 12-4. Multichannel Mode Configuration

SPORTx_RCR1 or SPORTx_RCR2	SPORTx_TCR1 or SPORTx_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Ignored
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0
RCKFE	TCKFE	Set or clear both to same value
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFSx signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFSx as a frame sync. This is true whether RFSx is generated internally or externally. The RFSx signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFSx indicates the beginning of the channel 0 data word.

Since RFSx is used by both the SPORTx_TX and SPORTx_RX channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in SPORTx_RCR1 and SPORTx_TCR1, and in SPORTx_RCR2 and SPORTx_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORTx_RX operation is not enabled.

In multichannel mode, RFSx timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFSx signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFSx signal serves as an output-enabled signal for the data transmit pin. The SPORT drives TFSx in multichannel mode whether or not ITFS is cleared. The TFSx pin in multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFSx pin indicates an active channel.

Once the initial RFSx is received, and a frame transfer has started, all other RFSx signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFSx may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFSx signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers.

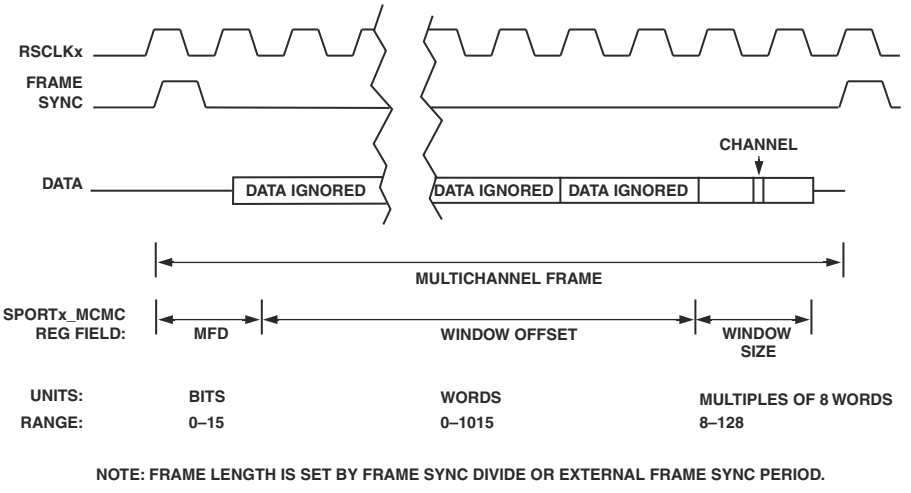


Figure 12-25. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit MFD field in SPORTx_MCMC2 specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (WSIZE[3:0]) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

```
Number of words in active window = 8 x (WSIZE + 1)
```

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

SPORT Current Channel (SPORTx_CHNL) Register

The 10-bit CHNL field in the SPORTx current channel register (SPORTx_CHNL) indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The CHNL[9:0] field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between RSCLKx and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind.

The addresses for these SPORT registers are:

SPORT0_CHNL – 0xFFC0 0834

SPORT1_CHNL – 0xFFC0 0934

SPORT2_CHNL – 0xFFC0 2534

SPORT3_CHNL – 0xFFC0 2634

SPORTx Current Channel Register (SPORTx_CHNL)
RO

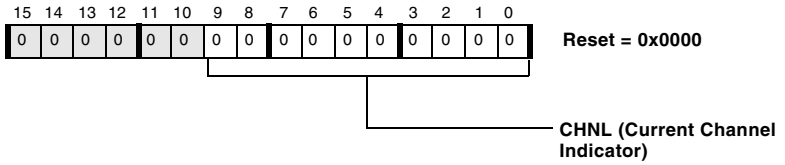


Figure 12-26. SPORTx Current Channel Register

Other Multichannel Fields in SPORTx_MCMC2

The FSDR bit in the SPORTx_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (when FSDR = 0), the data is transmitted on the same edge that the TFSx is generated. For example, a positive edge on TFSx causes data to be transmitted on the positive edge of the TSCLKx—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a 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).

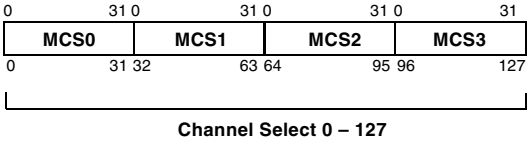


Figure 12-27. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in `MCS2` selects word 71 of the active window to be enabled. Setting bit 2 in `MCS1` selects word 34 of the active window, and so on.

Setting a particular bit in the `SPORTx_MTCSn` register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the `SPORTx_MTCSn` register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the `SPORTx_MRCSn` register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the `SPORTx_RX` buffer. Clearing the bit in the `SPORTx_MRCSn` register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the `TDTYPE` field in the `SPORTx_TCR1` register and the `RDTYPE` field in the `SPORTx_RCR1` register, and applies to all active channels. (See "Companding" on page 12-35 for more information about companding.)

SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers

The multichannel selection registers are used to enable and disable individual channels. The SPORTx multichannel receive select registers (SPORTx_MRCSn, see Figure 12-28 and Table 12-5) specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORTx_MRCSn register causes the serial port to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the serial port to ignore the data.

SPORTx Multichannel Receive Select Registers (SPORTx_MRCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

For memory-mapped addresses, see Table 12-5.

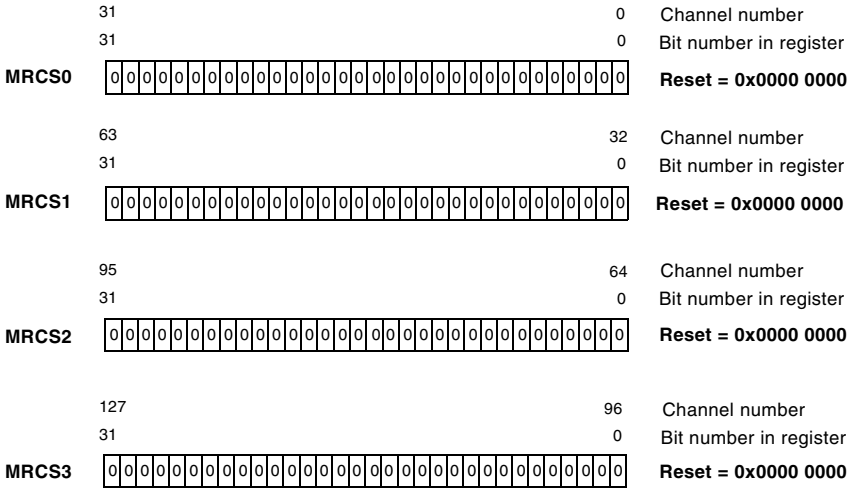


Figure 12-28. SPORTx Multichannel Receive Select Registers

Table 12-5. SPORTx Multichannel Receive Select Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address	Register Name	Memory-Mapped Address
SPORT0_MRCS0	0xFFC0 0850	SPORT2_MRCS0	0xFFC0 2550
SPORT0_MRCS1	0xFFC0 0854	SPORT2_MRCS1	0xFFC0 2554
SPORT0_MRCS2	0xFFC0 0858	SPORT2_MRCS2	0xFFC0 2558
SPORT0_MRCS3	0xFFC0 085C	SPORT2_MRCS3	0xFFC0 255C
SPORT1_MRCS0	0xFFC0 0950	SPORT3_MRCS0	0xFFC0 2650
SPORT1_MRCS1	0xFFC0 0954	SPORT3_MRCS1	0xFFC0 2654
SPORT1_MRCS2	0xFFC0 0958	SPORT3_MRCS2	0xFFC0 2658
SPORT1_MRCS3	0xFFC0 095C	SPORT3_MRCS3	0xFFC0 265C

SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers

The multichannel selection registers are used to enable and disable individual channels. The four SPORTx multichannel transmit select registers (`SPORTx_MTCSn`, see Figure 12-29 and Table 12-6) specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a `SPORTx_MTCSn` register causes the serial port to transmit the word in that channel's position of the datastream. Clearing the bit in the `SPORTx_MTCSn` register causes the serial port's data transmit pin to three-state during the time slot of that channel.

SPORTx Multichannel Transmit Select Registers (SPORTx_MTCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.
For memory-mapped addresses, see Table 12-6.

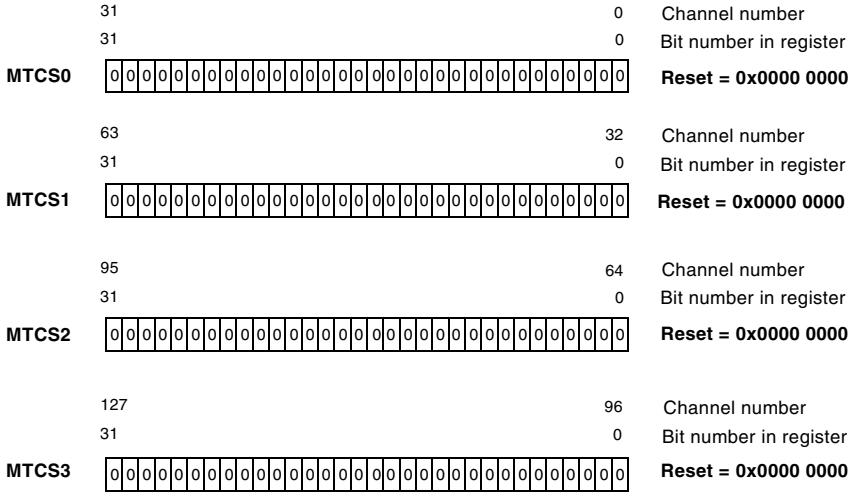


Figure 12-29. SPORTx Multichannel Transmit Select Registers

Table 12-6. SPORTx Multichannel Transmit Select Register Memory-mapped Addresses

Register Name	Memory-mapped Address	Register Name	Memory-mapped Address
SPORT0_MTCS0	0xFFC0 0840	SPORT2_MTCS0	0xFFC0 2540
SPORT0_MTCS1	0xFFC0 0844	SPORT2_MTCS1	0xFFC0 2544
SPORT0_MTCS2	0xFFC0 0848	SPORT2_MTCS2	0xFFC0 2548
SPORT0_MTCS3	0xFFC0 084C	SPORT2_MTCS3	0xFFC0 254C
SPORT1_MTCS0	0xFFC0 0940	SPORT3_MTCS0	0xFFC0 2640
SPORT1_MTCS1	0xFFC0 0944	SPORT3_MTCS1	0xFFC0 2644
SPORT1_MTCS2	0xFFC0 0948	SPORT3_MTCS2	0xFFC0 2648
SPORT1_MTCS3	0xFFC0 094C	SPORT3_MTCS3	0xFFC0 264C

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORTx_MCMC2 Multichannel Configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguring is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words. The data to be transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the Channel register to make sure that the active window is not being serviced. If the channel count is 0, then the Multichannel Select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- TFSR/RFSR set (frame syncs required)
- LTFS/LRFS set (active low frame syncs)
- Set for external clock
- MCMEN set (multichannel mode selected)
- MFD = 0 (no frame delay between frame sync and first data bit)
- SLEN = 7 (8-bit words)
- FSDR = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2X Clock Recovery Control

The SPORTs can recover the data rate clock from a provided 2X input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the SPORTx_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 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).

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed,", "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 12-42, and "Frame Syncs in Multichannel Mode" on page 12-54). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (`SLEN = 3`). Framing signals are active high (`LRFS = 0` and `LTFS = 0`).

Figure 12-30 through Figure 12-35 show framing for receiving data.

In Figure 12-30 and Figure 12-31, 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).

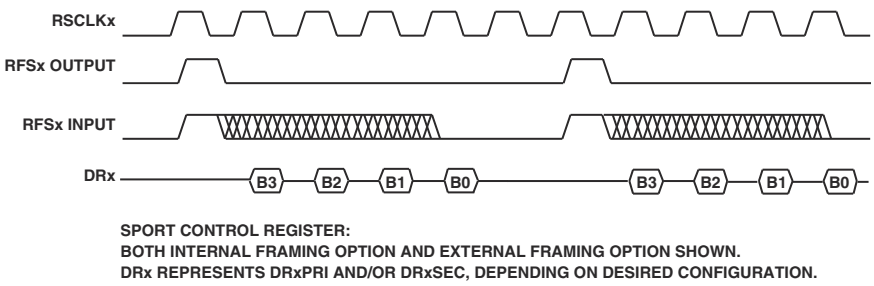


Figure 12-30. SPORT Receive, Normal Framing

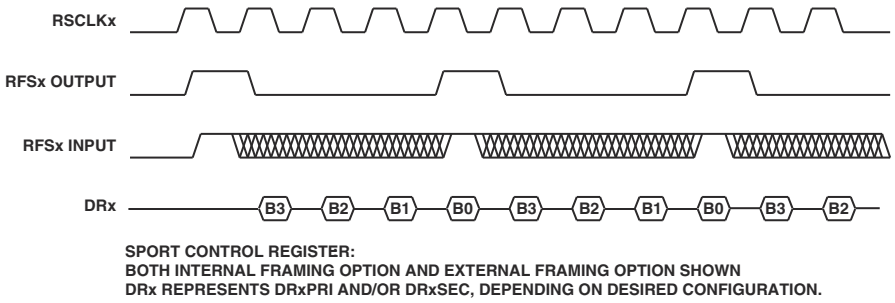


Figure 12-31. SPORT Continuous Receive, Normal Framing

Figure 12-32 and Figure 12-33 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.

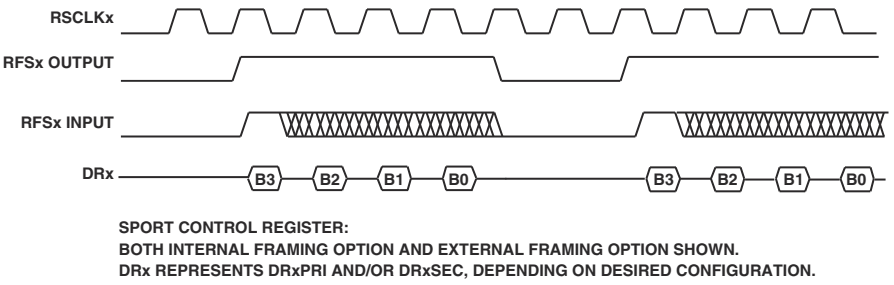


Figure 12-32. SPORT Receive, Alternate Framing

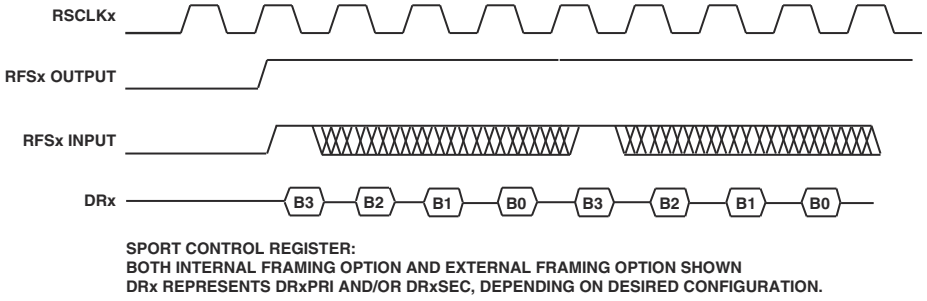


Figure 12-33. SPORT Continuous Receive, Alternate Framing

Figure 12-34 and Figure 12-35 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).

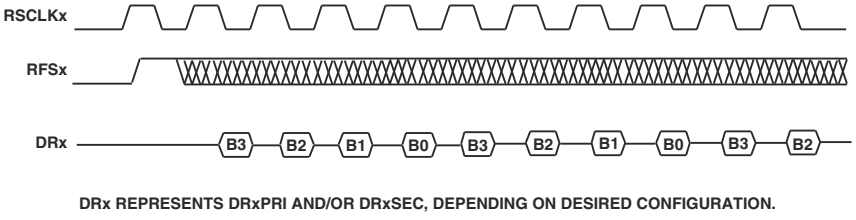


Figure 12-34. SPORT Receive, Unframed Mode, Normal Framing

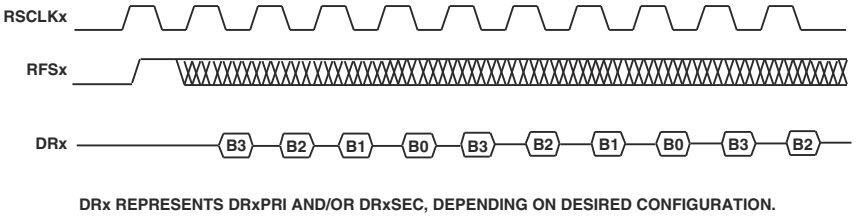


Figure 12-35. SPORT Receive, Unframed Mode, Alternate Framing

Figure 12-36 through Figure 12-41 show framing for transmitting data and are very similar to Figure 12-30 through Figure 12-35.

In Figure 12-36 and Figure 12-37, the normal framing mode is shown for non-continuous data (any number of TSCLKx cycles between words) and continuous data (no TSCLKx cycles between words). Figure 12-38 and Figure 12-39 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFSx output meets the RFSx input timing requirement.

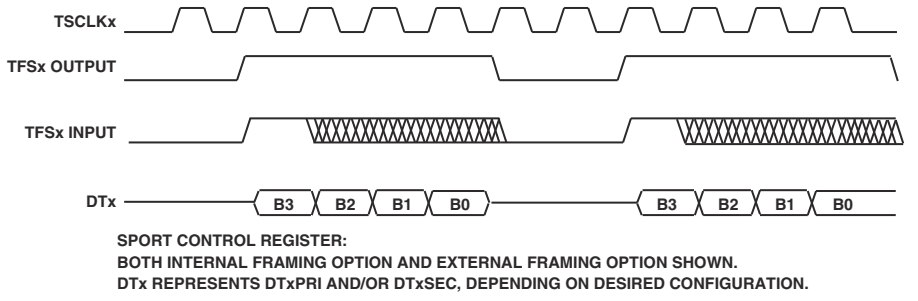


Figure 12-36. SPORT Transmit, Normal Framing

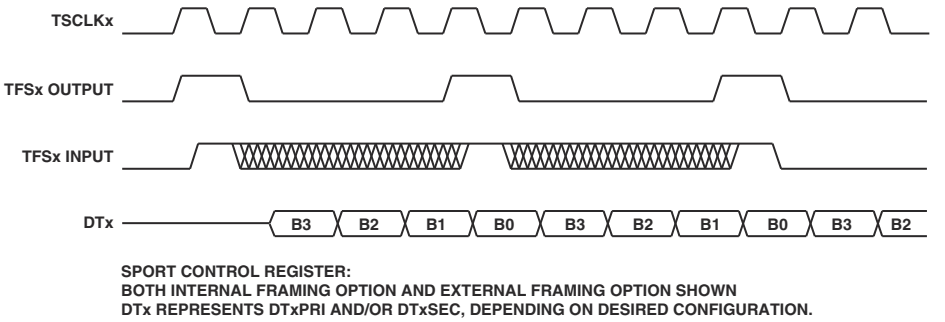


Figure 12-37. SPORT Continuous Transmit, Normal Framing

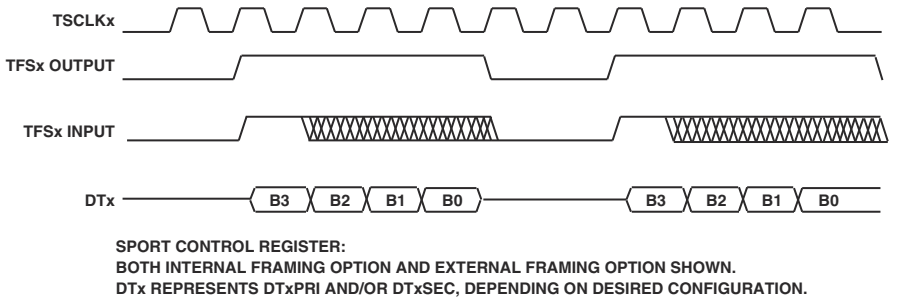


Figure 12-38. SPORT Transmit, Alternate Framing

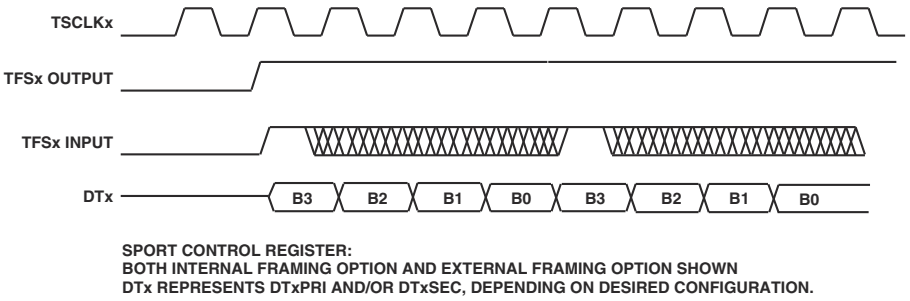


Figure 12-39. SPORT Continuous Transmit, Alternate Framing

Figure 12-40 and Figure 12-41 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

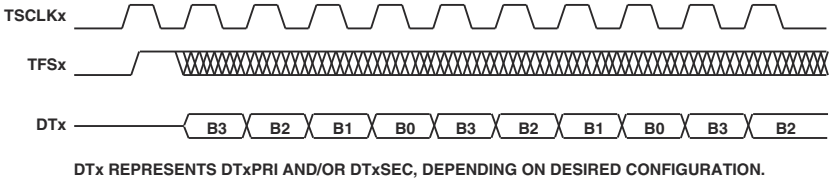


Figure 12-40. SPORT Transmit, Unframed Mode, Normal Framing

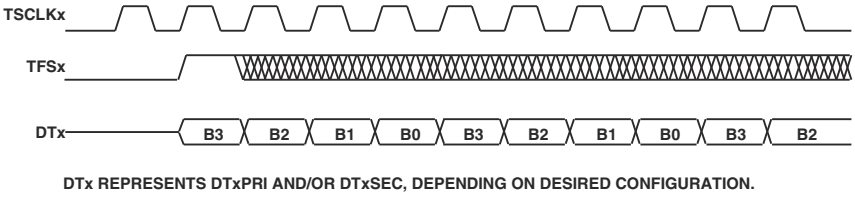


Figure 12-41. SPORT Transmit, Unframed Mode, Alternate Framing

13 UART PORT CONTROLLERS

The Universal Asynchronous Receiver/Transmitters (UART) are full-duplex peripherals, compatible with PC-style industry-standard UARTs. The ADSP-BF539 has three UARTs. Each UART converts data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, and parity generation options. Each UART includes interrupt handling hardware. Interrupts can be generated from 12 different events.

Each UART supports the half-duplex IrDA® (Infrared Data Association) SIR (9.6/115.2 Kbps rate) protocol. This is a mode-enabled feature.

Modem status and control functionality is not supported by the UART modules, but may be implemented using General-Purpose I/O (GPIO) pins.

Each UART is a DMA-capable peripheral with support for separate TX and RX DMA master channels. It can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. See Chapter 9, “Direct Memory Access” for more information on DMA.

UART0 has dedicated pins, but UART1 and UART2 have pins that can be used as GPIO. See Chapter 15, “General Purpose Input/Output Ports”.

For UART0, either of the peripheral timers can be used to provide a hardware assisted autobaud detection mechanism. See Chapter 16, “Timers” for more information.

UART1 and UART2 do not provide autobaud capabilities.

Serial Communications

Each UART follows an asynchronous serial communication protocol with these options:

- 5 – 8 data bits
- 1, 1½, or 2 stop bits
- None, even, or odd parity
- Baud rate = `SCLK/(16 × Divisor)`, where `SCLK` is the system clock frequency and `Divisor` can be a value from 1 to 65536, programmed in the `UARTx_DLH` and `UARTx_DLL` registers

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the Line control register (`UARTx_LCR`). Data is always transmitted and received least significant bit (LSB) first.

Figure 13-1 shows a typical physical bitstream measured on the TX pin:

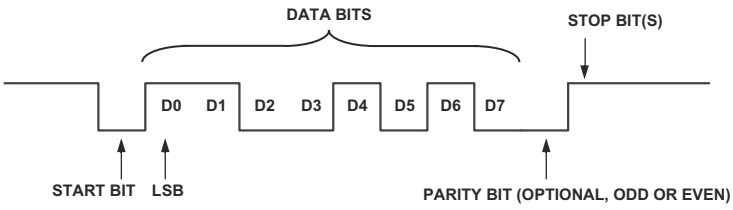


Figure 13-1. Bitstream on the TXx Pin

UART Control and Status Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These memory-mapped Registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled.

Consistent with industry-standard interfaces, multiple registers are mapped to the same address location. The divisor latch registers (UARTx_DLH and UARTx_DLL) share their addresses with the transmit holding register (UARTx_THR), the receive buffer register (UARTx_RBR), and the interrupt Enable register (UARTx_IER). The divisor latch access bit (DLAB) in the line control register (UARTx_LCR) controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UARTx_THR register buffers the transmit shift register (TSR) and the UARTx_RBR register buffers the receive shift register (LSR). The shift registers are not directly accessible by software.

UART Line Control (UARTx_LCR) Register

The UART line control register (UARTx_LCR) controls the format of received and transmitted character frames. The SB bit functions even when the UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

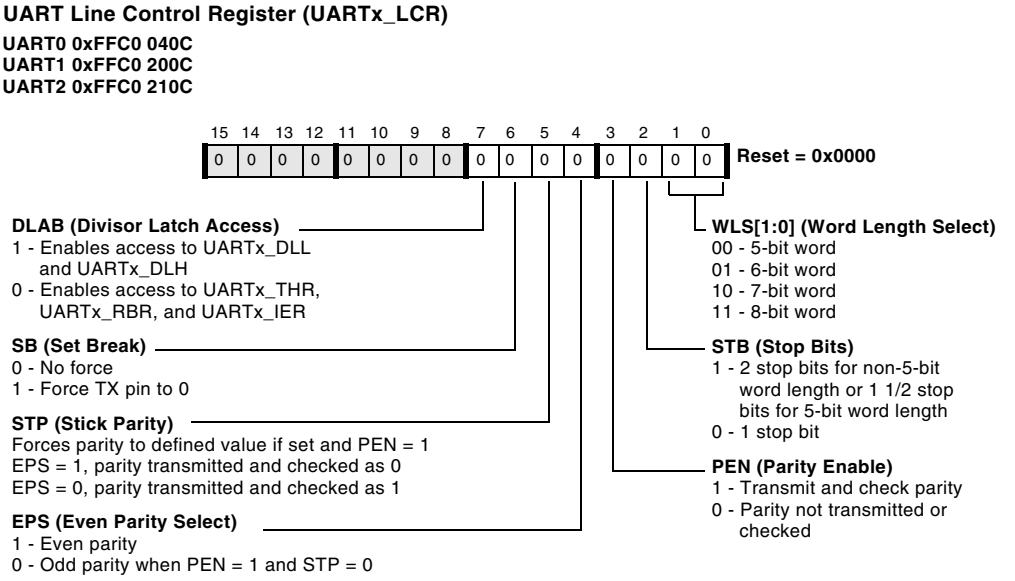


Figure 13-2. UART Line Control Register

UART Modem Control (UARTx_MCR) Register

The modem control register (UARTx_MCR) controls the UART port, as shown in Figure 13-3. Even if modem functionality is not supported, the

Modem control register is available in order to support the loopback mode.

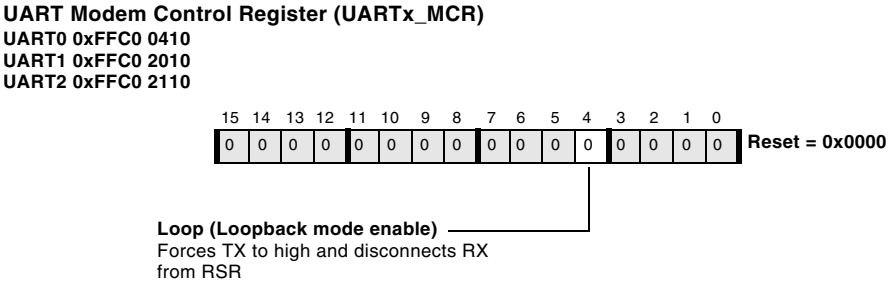


Figure 13-3. UART Modem Control Register

Loopback mode forces the TX pin to high and disconnects the receiver's input from the RX pin, but redirects it to the transmit output internally.

UART Line Status (UARTx_LSR) Register

The UART line status register (UARTx_LSR) contains UART status information as shown in Figure 13-4.

The break interrupt (BI), overrun error (OE), parity error (PE) and framing error (FE) bits are cleared when the UART line status register (UARTx_LSR) is read. The Data Ready (DR) bit is cleared when the UART receive buffer register (UARTx_RBR) is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see "Speculative Load Execution" on page 6-67 and "Conditional Load Behavior" on page 6-68.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UARTx_THR. Writes to UARTx_THR clear the THRE bit. It is set again when data is copied from UARTx_THR to the trans-

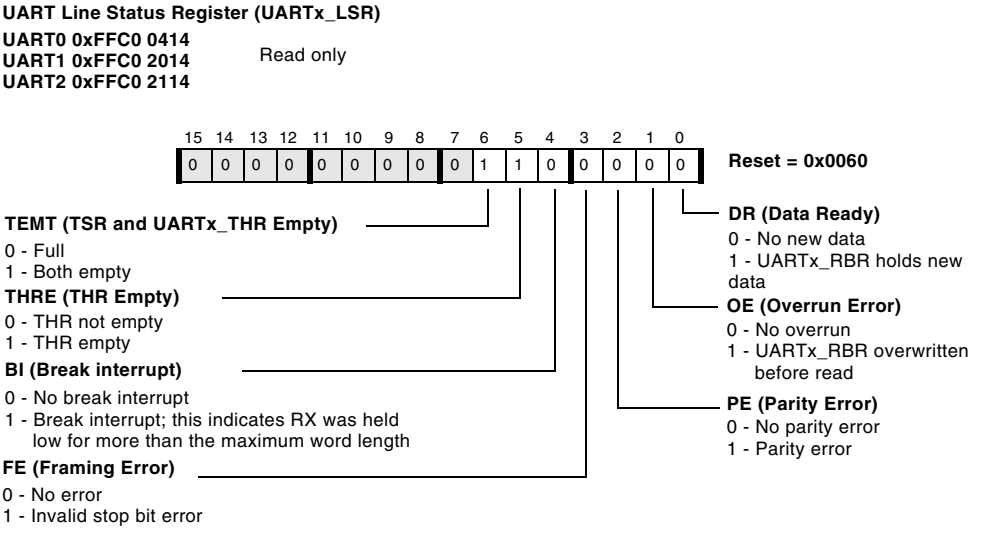


Figure 13-4. UART Line Status Register

mit shift register (TSR). The TEMT bit can be evaluated to determine whether a recently initiated transmit operation has been completed.

UART Transmit Holding (UARTx_THR) Register

A write to the UART transmit holding register (UARTx_THR) initiates the transmit operation. The data is moved to the internal transmit shift register (TSR) where it is shifted out at a baud rate equal to SCLK/(16 × Divisor) with start, stop, and parity bits appended as required. All data words begin with a 1-to-0-transition start bit. The transfer of data from UARTx_THR to the transmit shift register sets the Transmit Holding register Empty (THRE) status flag in the UART Line status register (UARTx_LSR).

The write-only UARTx_THR register is mapped to the same address as the read-only UARTx_RBR and UARTx_DLL registers. To access UARTx_THR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared,

writes to this address target the UARTx_THR register, and reads from this address return the UARTx_RBR register.

Note data is transmitted and received least significant bit (LSB) first (bit 0) followed by the most significant bits (MSBs).

UART Transmit Holding Register (UARTx_THR)

UART0 0xFFC0 0400
UART1 0xFFC0 2000 Write only
UART2 0xFFC0 2100

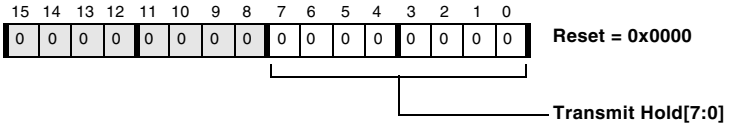


Figure 13-5. UART Transmit Holding Register

UART Receive Buffer (UARTx_RBR) Register

The receive operation uses the same data format as the transmit configuration, except that the number of stop bits is always assumed to be 1. After detection of the start bit, the received word is shifted into the receive shift register (RSR) at a baud rate of SCLK/(16 x Divisor). After the appropriate number of bits (including stop bit) is received, the data and any status are updated and the receive shift register is transferred to the UART receive buffer register (UARTx_RBR), shown in Figure 13-6. After the transfer of the received word to the UARTx_RBR buffer and the appropriate synchronization delay, the data ready (DR) status flag is updated.

A sampling clock equal to 16 times the baud rate samples the data as close to the midpoint of the bit as possible. Because the internal sample clock may not exactly match the asynchronous receive data rate, the sampling point drifts from the center of each bit. The sampling point is synchronized again with each start bit, so the error accumulates only over the

length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

The read-only UARTx_RBR register is mapped to the same address as the write-only UARTx_THR and UARTx_DLL registers. To access UARTx_RBR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, while reads from this address return the UARTx_RBR register.

UART Receive Buffer Register (UARTx_RBR)

UART0 0xFFC0 0400
UART1 0xFFC0 2000
UART2 0xFFC0 2100

Read only

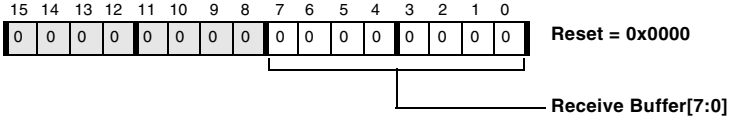


Figure 13-6. UART Receive Buffer Register

UART Interrupt Enable (UARTx_IER) Register

The UART interrupt enable register (UARTx_IER) is used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present. For backward compatibility, the UARTx_IIR still reflects the correct interrupt status.

The UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless of whether DMA is enabled or not.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UARTx_IER register is mapped to the same address as UARTx_DLH. To access UARTx_IER, the DLAB bit in UARTx_LCR must be cleared.

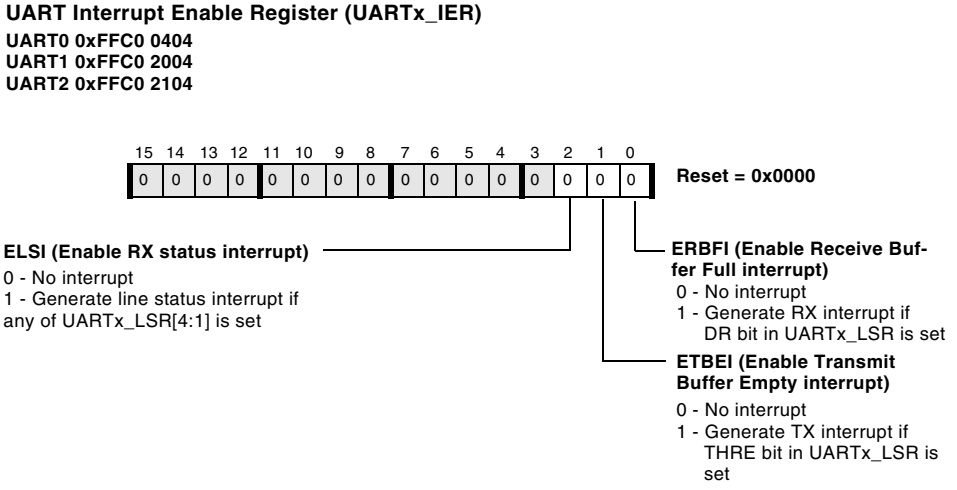


Figure 13-7. UART Interrupt Enable Register

The UART DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UARTx_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, the UART error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The ELSI bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UART Line status register (UARTx_LSR):

- Receive Overrun Error (OE)
- Receive Parity Error (PE)
- Receive Framing Error (FE)
- Break interrupt (BI)

When the ETBEI bit is set in the UARTx_IER register, the UART module immediately issues an interrupt or DMA request. When initiating the transmission of a string, no special handling of the first character is required. Set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UARTx_THR register in the normal manner. Accordingly, the ETBEI bit should be cleared if the string transmission has completed.

UART Interrupt Identification (UARTx_IIR) Register

For legacy reasons, the UART interrupt identification register (UARTx_IIR) still reflects the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the system interrupt controller (SIC).

When cleared, the pending interrupt bit (NINT) signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt. The receive line status has the highest priority; the UARTx_THR empty interrupt has the lowest priority. In the case where both interrupts are signalling, the UARTx_IIR reads 0x06.

When a UART interrupt is pending, the interrupt service routine (ISR) needs to clear the interrupt latch explicitly. The following figure shows how to clear any of the three latches.

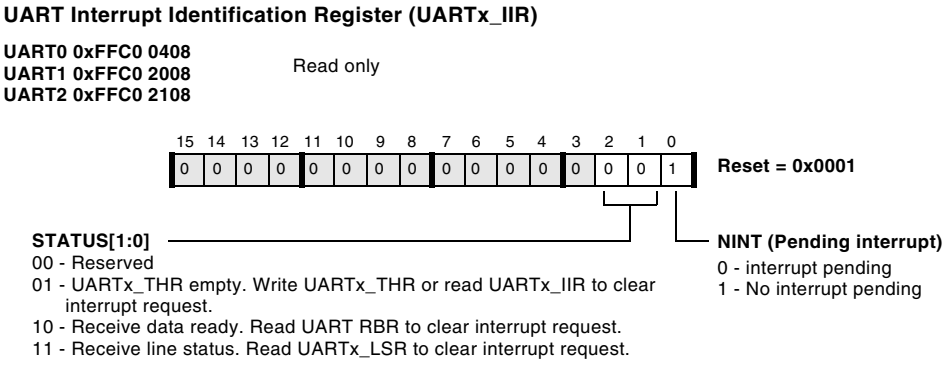


Figure 13-8. UART Interrupt Identification Register

The TX interrupt request is cleared by writing new data to the UARTx_THR register or by reading the UARTx_IIR register. Note the special role of the UARTx_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UARTx_IIR register to reset the interrupt request. As long as the UARTx_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UARTx_THR empty latch cannot be cleared by reading UARTx_IIR.

If either the line status interrupt or the receive data interrupt has been assigned a lower interrupt priority by the SIC, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UARTx_THR empty event.

Because of the destructive nature of these read operations, special care should be taken. For more information, see "Speculative Load Execution" on page 6-67 and "Conditional Load Behavior" on page 6-68.

UARTx_DLL and UARTx_DLH Registers

The bit rate is characterized by the system clock (SCLK) and the 16-bit Divisor. The divisor is split into the UART divisor latch low byte register (UART_DLL) and the UART divisor latch high byte register (UARTx_DLH). These registers form a 16-bit Divisor. The baud clock is divided by 16 so that:

```
BAUD RATE = SCLK/(16 x Divisor)
```

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

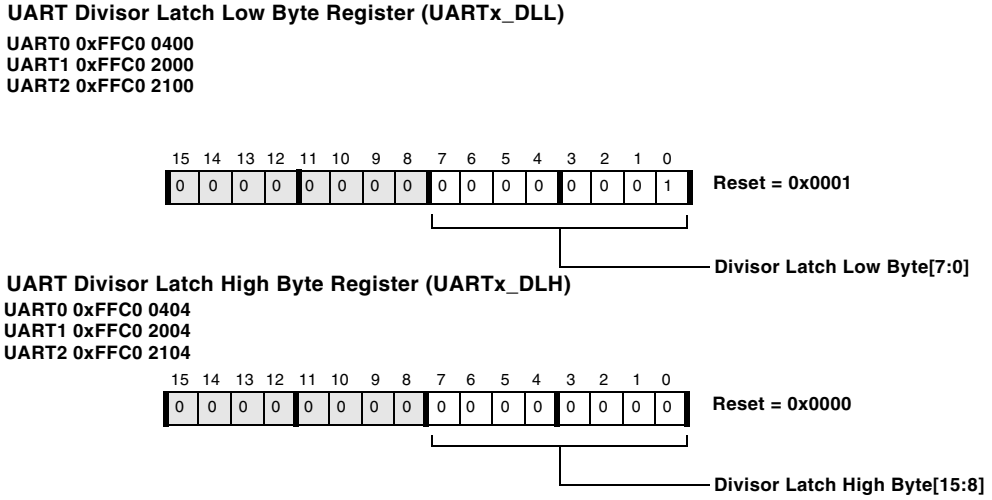


Figure 13-9. UART Divisor Latch Registers

The UART_DLL register is mapped to the same address as the UARTx_THR and UARTx_RBR registers. The UARTx_DLH register is mapped to the same address as the interrupt Enable register (UARTx_IER). The DLAB bit in UARTx_LCR must be set before the UART Divisor Latch registers can be accessed.

Note the 16-bit Divisor formed by UARTx_DLH and UARTx_DLL resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock will save power. The UARTx_DLH and UARTx_DLL registers can be programmed by software before or after setting the UCEN bit.

Table 13-1 provides example divide factors required to support most standard baud rates.

Table 13-1. UART Baud Rate Examples With 100 MHz SCLK

Baud Rate	DL	Actual	% Error
2400	2604	2400.15	.006
4800	1302	4800.31	.007
9600	651	9600.61	.006
19200	326	19171.78	.147
38400	163	38343.56	.147
57600	109	57339.45	.452
115200	54	115740.74	.469
921600	7	892857.14	3.119
6250000	1	6250000	-

Careful selection of SCLK frequencies, that is, even multiples of desired baud rates, can result in lower error percentages.

UART Scratch (UARTx_SCR) Register

The contents of the 8-bit UART scratch register (UARTx_SCR) is reset to 0x00. It is used for general-purpose data storage and does not control the UART hardware in any way.

UART Scratch Register (UARTx_SCR)

UART0 0xFFC0 041C
UART1 0xFFC0 201C
UART2 0xFFC0 211C

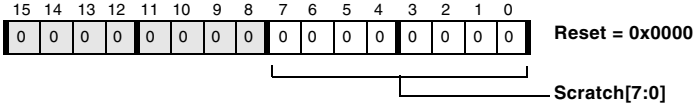


Figure 13-10. UART Scratch Register

UART Global Control (UARTx_GCTL) Register

The UART global control register (UARTx_GCTL) contains the enable bit for internal UART clocks and for the IrDA mode of operation of the UART.

Note that the UCEN bit was not present in previous UART implementations. It has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

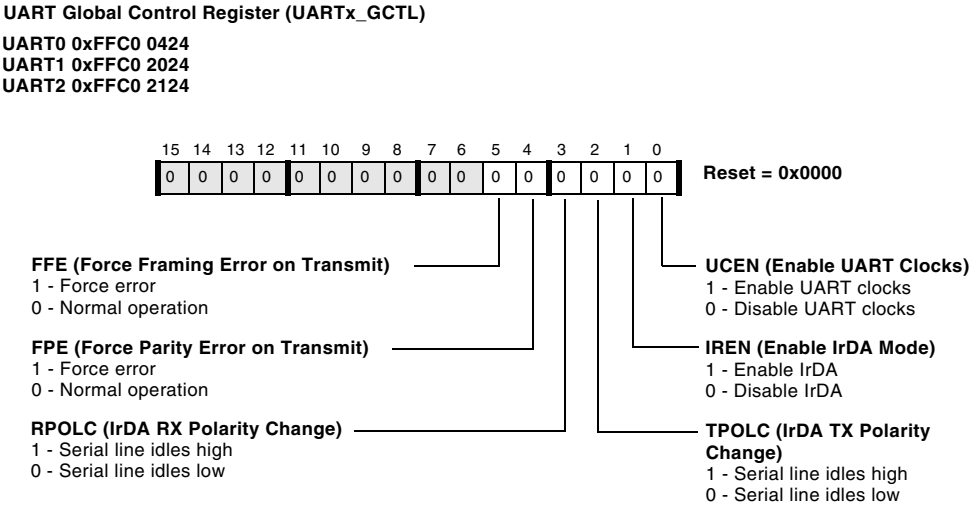


Figure 13-11. UART Global Control Register

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into UARTx_THR. Received data can be read from UARTx_RBR. The processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial datastream, the UART Line status register (UARTx_LSR) provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UARTx_THR is ready for new data and cleared when the processor loads new data into UARTx_THR. Writing UARTx_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UARTx_RBR. This flag is cleared automatically when the processor reads from UARTx_RBR. Reading UARTx_RBR when it is not full returns the previously received value. When UARTx_RBR is not read in time, newly received data overwrites UARTx_RBR and the overrun (OE) flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Software can write up to two words into the UARTx_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines (ISRs). Separate interrupt lines are provided for UART TX, UART RX, and UART Error. The independent interrupts can be enabled individually by the UARTx_IER register.

The ISRs can evaluate the status bit field within the UART interrupt Identification register (UARTx_IIR) to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 13-8 on page -11.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through Autobuffer mode.

No additional buffering is provided in the UART DMA channel, so the latency requirements are the same as in non-DMA mode. However, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities. For more information, see Chapter 9, “Direct Memory Access”.

DMA interrupt routines must explicitly write 1s to the corresponding DMA IRQ status registers to clear the latched request of the pending interrupt.

The UART DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UARTx_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, the UART error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The UART DMA supports 8-bit operation.

Mixing Modes

Non-DMA and DMA modes use different synchronization mechanisms. Consequently, any serial communication must be complete before switching from non-DMA to DMA mode or vice versa. In other words, before switching from non-DMA transmission to DMA transmission, make sure both UARTx_THR and the internal Transmit Shift register (TSR) are empty by testing the THRE and the TEMT status bits in UARTx_LSR. Otherwise, the processor must wait until the 2-bit DMA Buffer status field within the appropriate UART Transmit DMA Configuration register (UARTx_CONFIG_TX) is clear.

When switching from DMA to non-DMA operation, make sure both the receive (RX) and transmit (TX) DMA channels have completely transferred their data, including data contained in the DMA FIFOs. While the DMA RX interrupt indicates the last data word has been written to memory (and has left the DMA FIFO), the DMA TX interrupt indicates the last data word has left memory (and has entered the DMA FIFO). The processor must wait until the TX FIFO is empty, by testing that the `DMA_RUN` status bit in the TX channel's `IRQ_STATUS` register is clear, before it is safe to disable the DMA channel.

IrDA Support

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

IrDA support is enabled by setting the `IREN` bit in the UART global control register. The IrDA application requires external transceivers.

IrDA Transmitter Description

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3/16 clock periods in a 16-cycle UART clock period. The pulse is centered around the middle of the bit time, as shown in Figure 13-12. The final IrDA pulse is fed to the off-chip infrared driver.

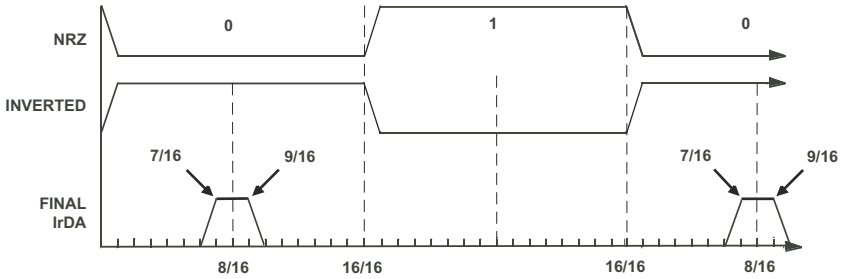


Figure 13-12. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 13-1 on page -13, the error terms associated with the baud rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receiver Description

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16x bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the `IRPOL` bit. Figure 13-13 gives examples of each polarity type.

- `IRPOL` = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- `IRPOL` = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

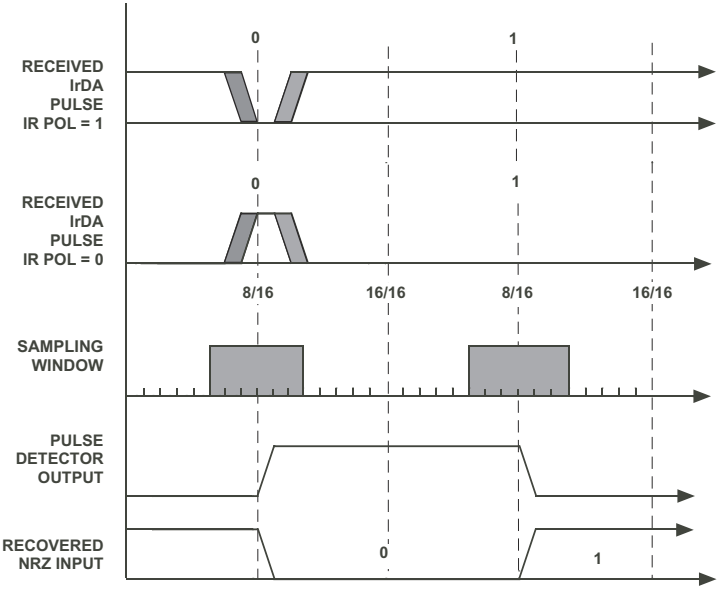


Figure 13-13. IrDA Receiver Pulse Detection

14 PROGRAMMABLE FLAGS

The processor supports 16 bidirectional programmable flags (PFx) PF[15:0]. Each pin can be individually configured as either an input or an output by using the flag direction register (FIO_DIR). When configured as output, the flag data register (FIO_FLAG_D) can be directly written to specify the state of all PFx pins. When configured as an output, the state written to the flag set (FIO_FLAG_S), flag clear (FIO_FLAG_C), and flag toggle (FIO_FLAG_T) registers determines the state driven by the output PFx pin. Regardless of whether the pins are configured, as inputs or outputs, reading any of these registers (FIO_FLAG_D, FIO_FLAG_S, FIO_FLAG_C, FIO_FLAG_T) returns the state of each pin.

Each PFx pin can be configured to generate an interrupt. When a PFx pin is configured as an input, an interrupt can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high *and* high to low). Input sensitivity is defined on a per-bit basis by the flag polarity register (FIO_POLAR), the flag interrupt sensitivity register (FIO_EDGE) and the flag set on both edges register (FIO_BOTH). Input polarity is defined on a per-bit basis by the FIO_POLAR register. When the PFx inputs are enabled and a PFx pin is configured as an output, enabling interrupts for the pin allows an interrupt to be generated by setting the PFx pin.

The processor provides two independent interrupt channels for the PFx pins. Identical in functionality, these are called interrupt A and interrupt B. Each interrupt channel has four mask registers associated with it, a flag interrupt mask data register (FIO_MASKx_D), a flag interrupt mask set register (FIO_MASKx_S), a flag interrupt mask clear register (FIO_MASKx_C), and a flag interrupt mask toggle register (FIO_MASKx_T).

Each PFx pin is represented by a bit in each of these eight registers. Writing a 1 to a bit in a FIO_MASKx_S register enables interrupt generation for that PFx pin, while writing a 1 to a bit in a FIO_MASKx_C register disables interrupt generation for that PFx pin.

The interrupt masking can be toggled by writing a 1 to a bit in the FIO_MASKx_T register. Additionally, the mask bits can be directly written by writing to the FIO_MASKx_D register. This flexible mechanism allows each bit to generate flag interrupt A, flag interrupt B, both flag interrupts A and B, or neither.

When a PFx pin is not used in a system, the input buffer can be disabled so that no external pull-ups or pull-downs are required on the unused pins. By default, the input buffers are disabled. They can be enabled via bits in the Flag Input Enable register (FIO_INEN).

The PFx pins are multiplexed for use by the parallel peripheral interface (PPI), timers, and serial peripheral interface (SPI0). Table 14-1 shows the programmable flag pins and their multiplexed functionality.

Table 14-1. Programmable Flag Pins and Functionality

PF Pin	Peripheral That Shares the PF Pin		
	PPI	SPI	Timers 0, 1, 2
PF0		Slave Select Input ($\overline{\text{SPIOSS}}$)	
PF1		Slave Select Enable 1 ($\overline{\text{SPIOSEL1}}$)	Input clock (TMRCLK)
PF2		Slave Select Enable 2 ($\overline{\text{SPIOSEL2}}$)	
PF3	Frame Sync 3 (PPI_FS3)	Slave Select Enable 3 ($\overline{\text{SPIOSEL3}}$)	
PF4	I/O #15 (PPI15)	Slave Select Enable 4 ($\overline{\text{SPIOSEL4}}$)	
PF5	I/O #14 (PPI14)	Slave Select Enable 5 ($\overline{\text{SPIOSEL5}}$)	
PF6	I/O #13 (PPI13)	Slave Select Enable 6 ($\overline{\text{SPIOSEL6}}$)	
PF7	I/O #12 (PPI12)	Slave Select Enable 7 ($\overline{\text{SPIOSEL7}}$)	
PF8	I/O #11 (PPI11)		
PF9	I/O #10 (PPI10)		
PF10	I/O #9 (PPI9)		
PF11	I/O #8 (PPI8)		
PF12	I/O #7 (PPI7)		
PF13	I/O #6 (PPI6)		
PF14	I/O #5 (PPI5)		
PF15	I/O #4 (PPI4)		

Table 14-2 describes how to use the peripheral function that shares the PFx pins.

Table 14-2. How to Use Peripheral Function That Shares the PF Pin

PFx Pin	To Use the Peripheral Function That Shares the PFx Pin… (Assumes Peripheral is Enabled)		
	PPI	SPI	Timers 0, 1,2
0		Set PSSE in SPI0_CTL	
1		Set FLS1 in SPI0_FLG	Write 1 to CLK_SEL in TIMERx_CONFIG
2		Set FLS2 in SPI0_FLG	
3	In PPI_CTL: If PORT_DIR = 1, write b#01 to PORT_CFG. If PORT_DIR = 0, write b#10 to PORT_CFG.	Set FLS3 in SPI0_FLG	
4	Write b#111 to DLEN in PPI_CTL	Set FLS4 in SPI0_FLG	
5	Write b#110 to DLEN in PPI_CTL	Set FLS5 in SPI0_FLG	
6	Write b#101 to DLEN in PPI_CTL	Set FLS6 in SPI0_FLG	
7	Write b#100 to DLEN in PPI_CTL	Set FLS7 in SPI0_FLG	
8	Write b#011 to DLEN in PPI_CTL		
9	Write b#010 to DLEN in PPI_CTL		
10	Write b#001 to DLEN in PPI_CTL		
11	Write b#001 to DLEN in PPI_CTL		
12	Always enabled when PPI enabled		
13	Always enabled when PPI enabled		
14	Always enabled when PPI enabled		
15	Always enabled when PPI enabled		

For more information, see Chapter 11, “Parallel Peripheral Interface” Chapter 10, “SPI Compatible Port Controllers”and Chapter 16, “Timers”

Programmable Flag Registers (MMRs)

The programmable flag registers are part of the system memory-mapped registers (MMRs). The addresses of the programmable flag MMRs appear in Chapter B, "System MMR Assignments". Core access to the flag configuration registers is through the system bus.

Flag Direction (FIO_DIR) Register

The flag direction register (FIO_DIR) is a read-write register. Each bit position corresponds to a PFx pin. A logic 1 configures the PFx pin as an output, driving the state contained in the FIO_FLAG_D register. A logic 0 configures the PFx pin as an input. The reset value of this register is 0x0000, making all PFx pins inputs upon reset.

When using the PFx pin as an input, the corresponding bit should also be set in the flag input enable (FIO_INEN) register.

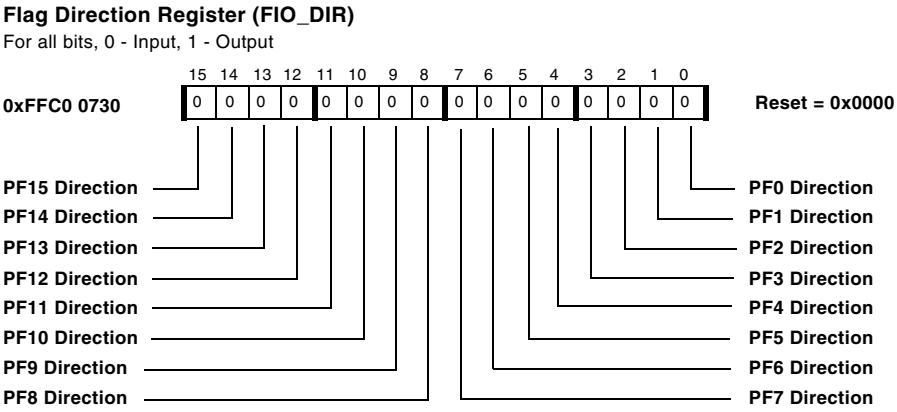


Figure 14-1. Flag Direction Register

Flag Value Registers Overview

The processor has four flag value registers:

- Flag data register (FIO_FLAG_D)
- Flag set register (FIO_FLAG_S)
- Flag clear register (FIO_FLAG_C)
- Flag toggle direct register (FIO_FLAG_T)

These registers are used to:

- Sense the value of the PFx pins defined as inputs
- Specify the state of PFx pins defined as outputs
- Clear interrupts generated by the PFx pins

Each PFx pin is represented by a bit in each of the four registers.

Reading any of the FIO_FLAG_D, FIO_FLAG_S, FIO_FLAG_C, or FIO_FLAG_T registers returns the value of the PFx pins. The value returned shows the state of the PFx pins defined as outputs and the sense of PFx pins defined as inputs, based on the polarity and sensitivity settings of each pin.

Reading the FIO_FLAG_D, FIO_FLAG_S, FIO_FLAG_C, or FIO_FLAG_T register after reset results in 0x0000 because although the pins are inputs, the input buffers are not enabled. See Table 14-3 for guidance on how to interpret a value read from one of these registers, based on the settings of the FIO_POLAR, FIO_EDGE, and FIO_BOTH registers.

Table 14-3. Flag Value Register Pin Interpretation

FIO_POLAR	FIO_EDGE	FIO_BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For pins configured as edge-sensitive, a read back of 1 from one of these registers is sticky. That is, once the bit is set, it remains set until cleared by the program. For level-sensitive pins, the pin state is checked every cycle, so the read back value changes when the level on the pin changes.

For more information about the flag set, flag clear, and flag toggle registers, see "Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) Registers" on page 14-8.

Flag Data (FIO_FLAG_D) Register

When written, the flag data register (FIO_FLAG_D), shown in Figure 14-2, directly specifies the state of all PFx pins. When read, the register returns the value of the PFx pins.

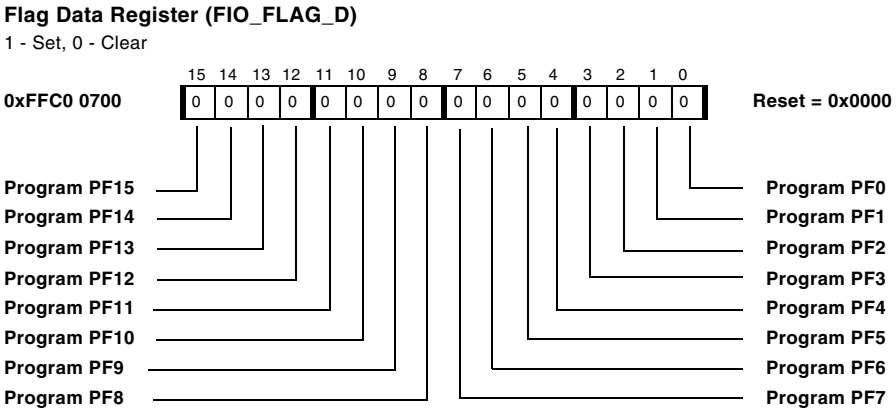


Figure 14-2. Flag Data Register

Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) Registers

The flag set (FIO_FLAG_S), flag clear (FIO_FLAG_C), and flag toggle (FIO_FLAG_T) registers are used to:

- Set, clear or toggle the output state associated with each output PFx pin
- Clear the latched interrupt state captured from each input PFx pin

This mechanism is used to avoid the potential issues with more traditional read-modify-write mechanisms. Reading any of the these registers returns the flag pin state.

Figure 14-3 and Figure 14-4 represent the FIO_FLAG_S and FIO_FLAG_C registers, respectively. Figure 14-5 represents the FIO_FLAG_T register.

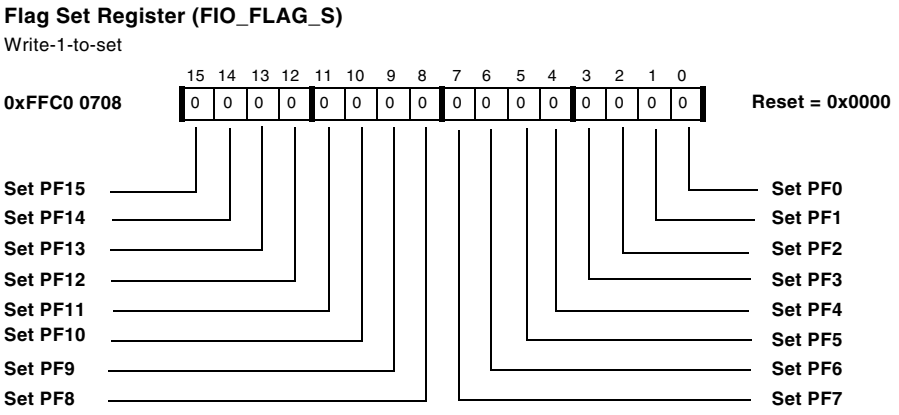


Figure 14-3. Flag Set Register

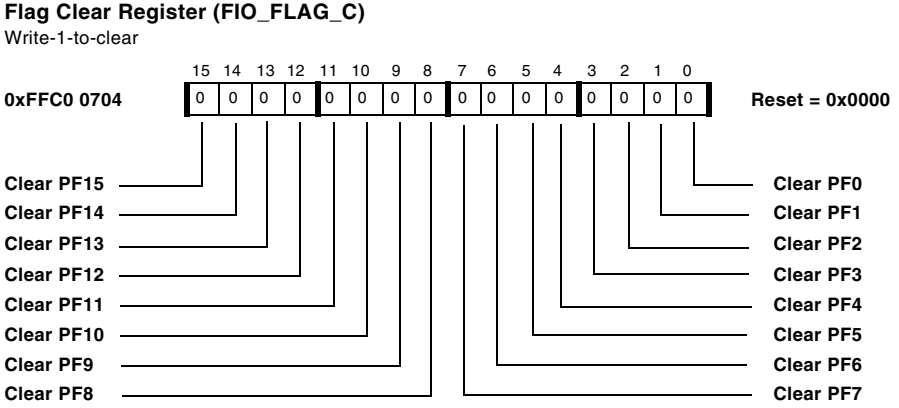


Figure 14-4. Flag Clear Register

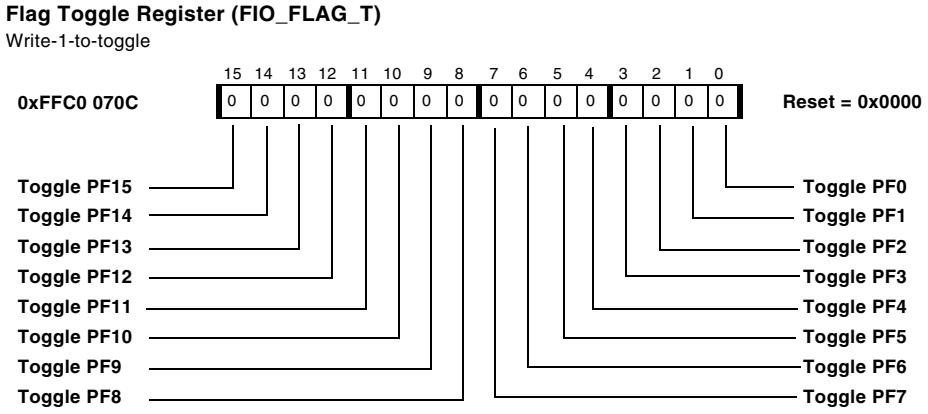


Figure 14-5. Flag Toggle Register

As an example of how these registers work, assume that PF0 is configured as an output (FIO_DIR = 0x0001). Writing 0x0001 to the FIO_FLAG_S register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. Writing 0x0001 to the FIO_FLAG_C register drives a logic 0 on the PF0 pin without affecting the state of any other PFx pins. Writing a 0x0001 to the FIO_FLAG_T register changes the pin state on PF0 from logic 0 to logic 1 or from logic 1 to logic 0, depending upon the existing pin state, without affecting the state of any other PFx pins.

Writing a 0 to the FIO_FLAG_S, FIO_FLAG_C, or FIO_FLAG_T registers has no effect on the value of the flag pin and is therefore ignored.

Reading the FIO_FLAG_S or FIO_FLAG_C registers returns:

- 0s for PFx pins defined as outputs and driven low
- 1s for pins (including PF0 in the example above) defined as outputs and driven high
- The present sense of PFx pins defined as inputs

Input sense is based on FIO_POLAR and FIO_EDGE register settings, as well as the logic level at each pin.

Flag Mask Interrupt Registers Overview

The processor supports up to two interrupt sources for each of the flag pins—flag interrupt A and flag interrupt B. These interrupts are configurable in a set of flag mask interrupt registers (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, and FIO_MASKB_T) which are implemented as complementary pairs of write-1-to-set, write-1-to-clear, and write-1-to-toggle registers.

Both flag interrupt A and flag interrupt B are supported by a set of four dedicated registers:

- Flag mask interrupt data registers (FIO_MASKA_D and FIO_MASKB_D)
- Flag mask interrupt set registers (FIO_MASKA_S and FIO_MASKB_S)
- Flag mask interrupt clear registers (FIO_MASKA_C and FIO_MASKB_C)
- Flag interrupt toggle registers (FIO_MASKA_T and FIO_MASKB_T)

This implementation provides the ability to enable or disable a PFx pin to act as a processor interrupt without requiring read-modify-write accesses—or to directly specify the mask value with the data register. For diagrams of the registers that support flag interrupt A, see "Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14.

For diagrams of the registers that support flag interrupt B, see "Flag Interrupt B (FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T) Registers" on page 14-16.

Each PFx pin is represented by a bit in each of the eight registers. Table 14-4 shows the effect of writing 1 to a bit in a FIO_MASKx_S, FIO_MASKx_C, or FIO_MASKx_T registers.

Table 14-4. Effect of Writing 1 to a Bit

Register	Effect of Writing 1 to a Bit in the Register
FIO_MASKx_S	Enables flag x interrupt generation for that PFx pin
FIO_MASKx_C	Disables flag x interrupt generation for that PFx pin
FIO_MASKx_T	Changes the state of flag x interrupt generation capability

Reading any of the FIO_MASKx_D, FIO_MASKx_S, FIO_MASKx_C, or FIO_MASKx_T registers returns the value of the current FIO_MASKx_D register.

Flag interrupt A and flag interrupt B operate independently. For example, writing 1 to a bit in the FIO_MASKA_S register does not affect flag interrupt B. This facility allows PFx pins to generate flag interrupt A, flag interrupt B, both flag interrupts A and B, or neither.

A flag interrupt is generated by a logical OR of all unmasked PFx pins for that interrupt. For example, if PF0 and PF1 are both unmasked for flag interrupt A, flag interrupt A is generated when triggered by PF0 or PF1.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate FIO_FLAG_C register bit. For level-triggered interrupts, this is not a requirement.

At reset, all interrupts are masked.

Flag Interrupt Generation Flow

Figure 14-6 shows the process by which flag interrupt A or flag interrupt B generates an event. Note the flow is shown for only one programmable flag, “FLAGn.” However, a flag interrupt is generated by a logical OR of all unmasked PFx pins for that interrupt. For example, if only PF0 and PF1 are unmasked for flag interrupt A, this interrupt is generated when triggered by either PF0 or PF1.

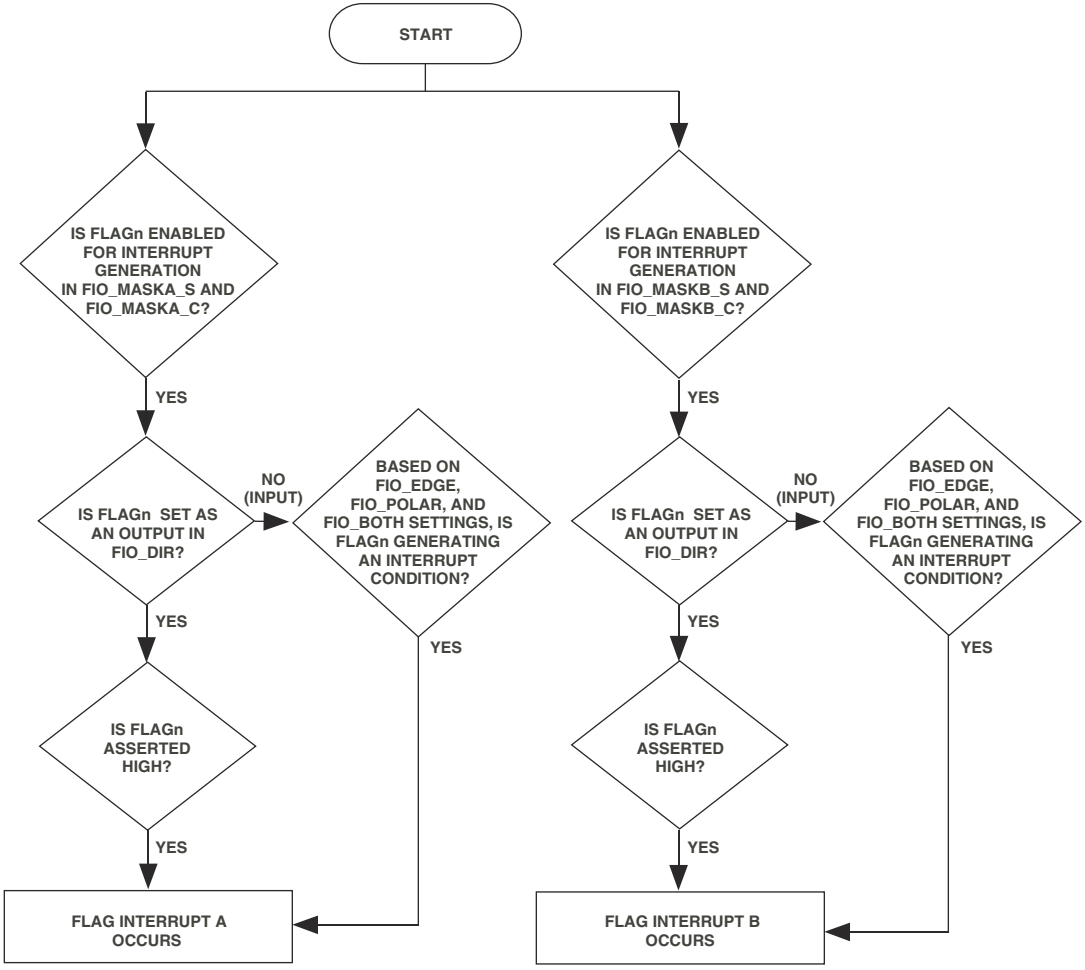


Figure 14-6. Flag Interrupt Generation Flow

Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers

The registers shown in Figure 14-7 through Figure 14-10 support flag interrupt A. For details, see page -11.

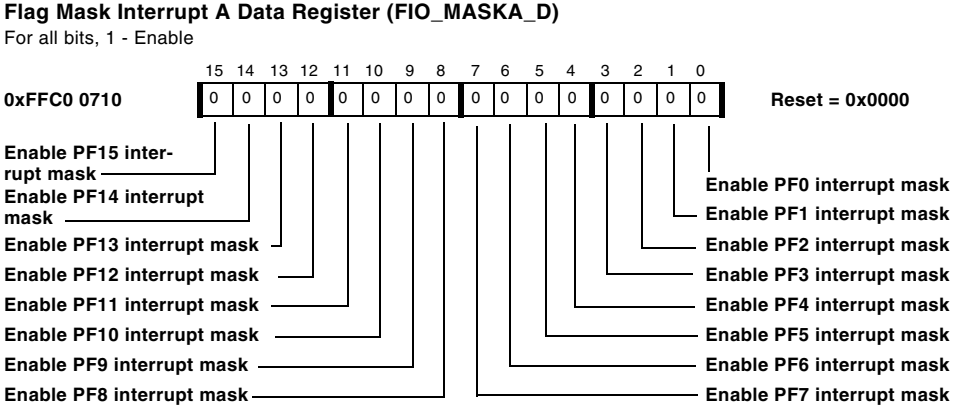


Figure 14-7. Flag Mask Interrupt A Data Register

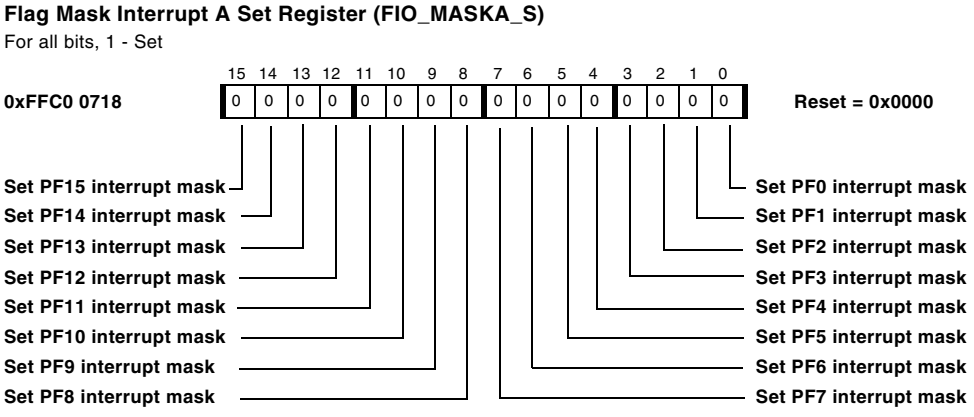


Figure 14-8. Flag Mask Interrupt A Set Register

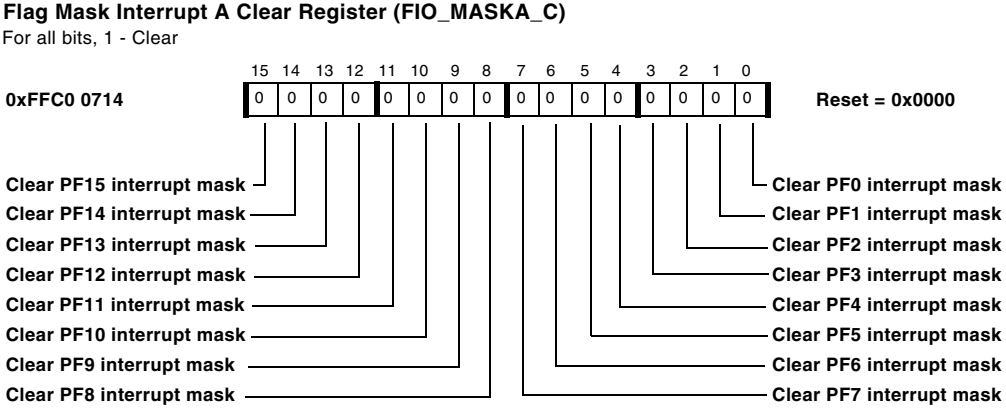


Figure 14-9. Flag Mask Interrupt A Clear Register

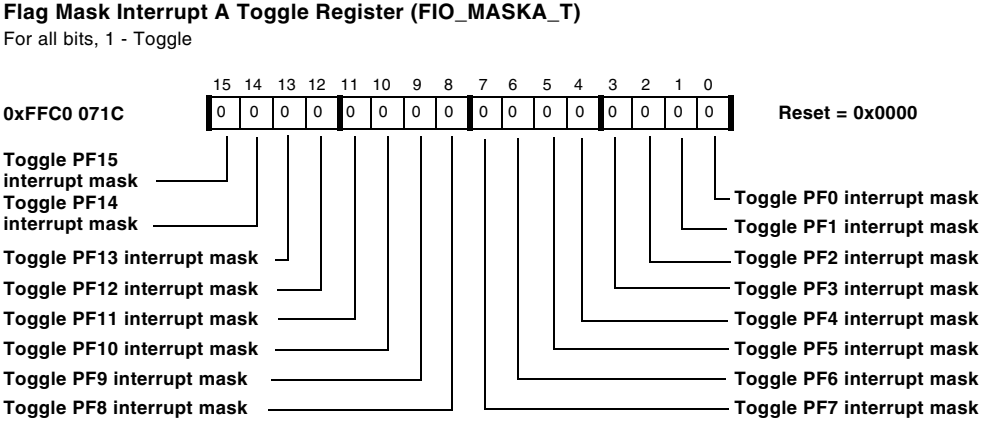


Figure 14-10. Flag Mask Interrupt A Toggle Register

Flag Interrupt B (FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T) Registers

The shown in Figure 14-11 through Figure 14-14 support flag interrupt B. For details, see page -11.

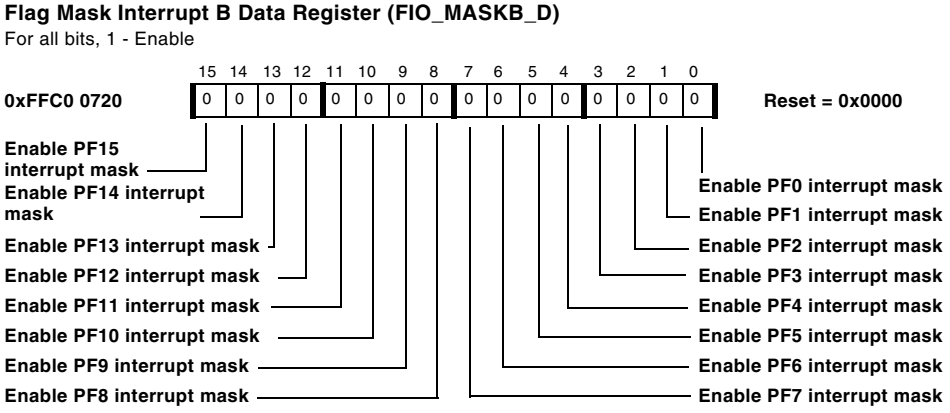


Figure 14-11. Flag Mask Interrupt B Data Register

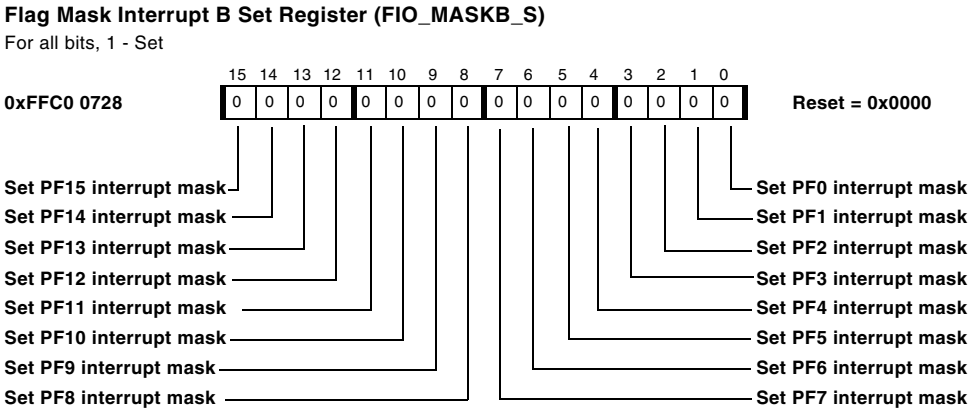


Figure 14-12. Flag Mask Interrupt B Set Register

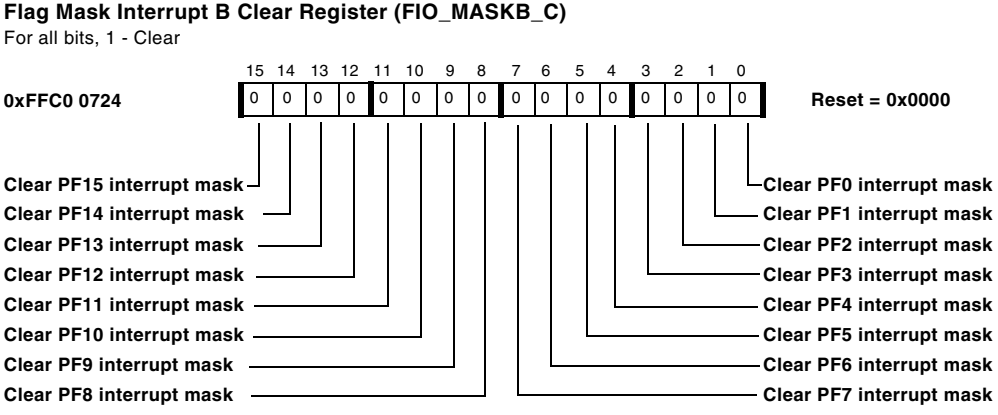


Figure 14-13. Flag Mask Interrupt B Clear Register

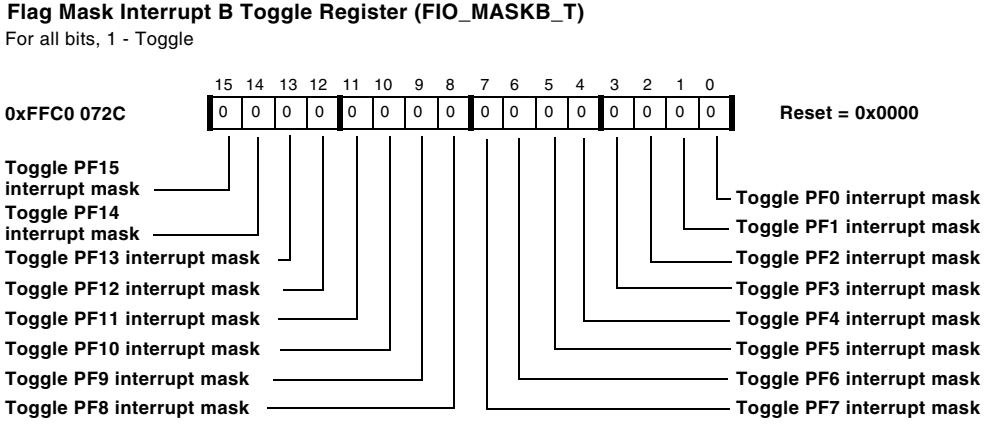


Figure 14-14. Flag Mask Interrupt B Toggle Register

Flag Polarity (FIO_POLAR) Register

The flag polarity register (FIO_POLAR) shown in Figure 14-15 is used to configure the polarity of the flag input source. To select active high or rising edge, set the bits in this register to 0. To select active low or falling edge, set the bits in this register to 1.

This register has no effect on PFx pins that are defined as outputs. The contents of this register are cleared at reset, defaulting to active high polarity.

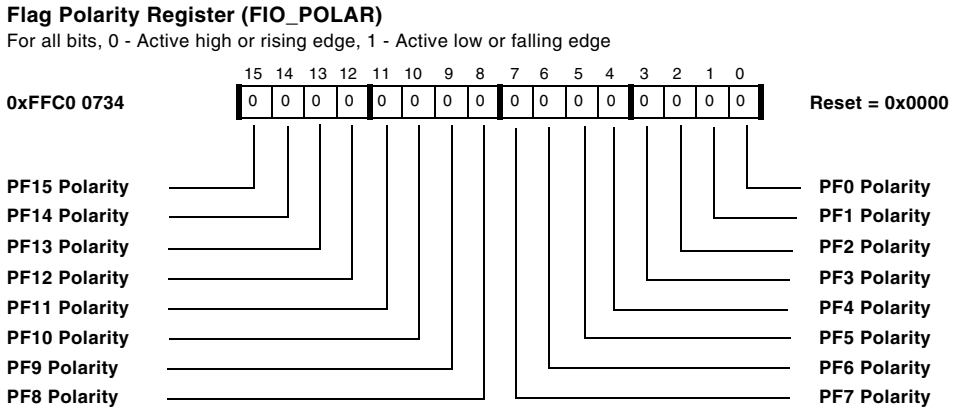


Figure 14-15. Flag Polarity Register

Flag Interrupt Sensitivity (FIO_EDGE) Register

The flag interrupt sensitivity register (FIO_EDGE) shown in Figure 14-16 is used to configure each of the flags as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. This register has no effect on PFx pins that are defined as outputs.

The contents of this register are cleared at reset, defaulting to level sensitivity.

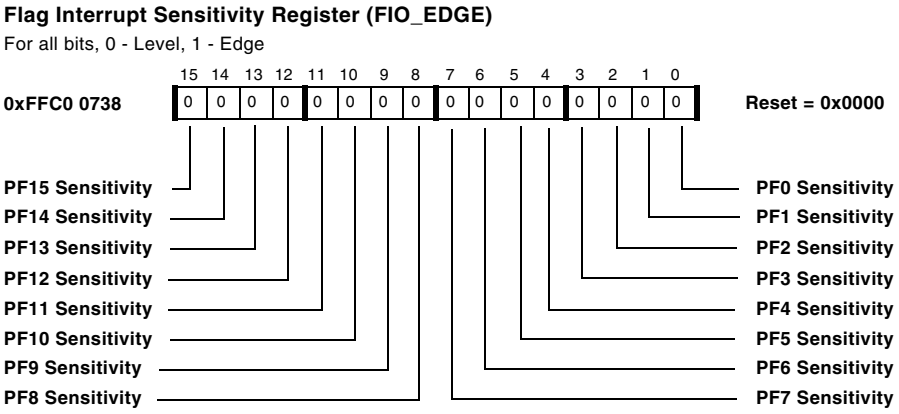


Figure 14-16. Flag Interrupt Sensitivity Register

Flag Set on Both Edges (FIO_BOTH) Register

The flag set on both edges register (FIO_BOTH) shown in Figure 14-17 is used to enable interrupt generation on both rising and falling edges.

When a given PFx pin has been set to edge-sensitive in the Flag interrupt sensitivity register, setting the PFx pin's bit in the flag set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on PFx pins that are defined as level-sensitive or as outputs.

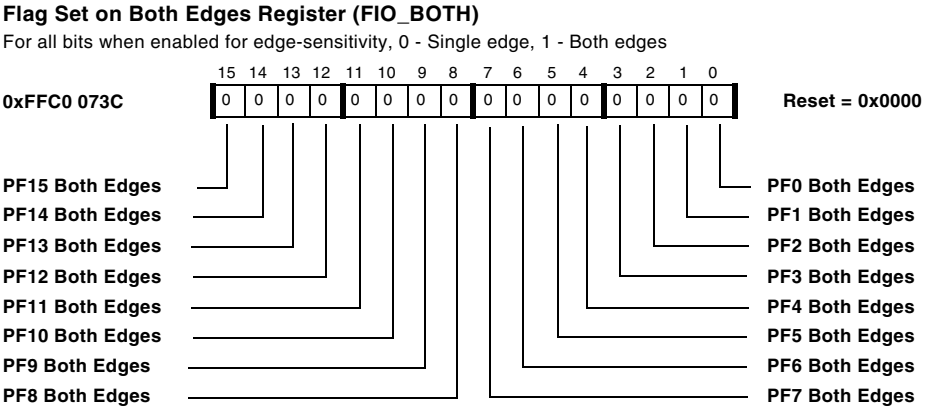


Figure 14-17. Flag Set on Both Edges Register

Flag Input Enable (FIO_INEN) Register

The flag input enable register (FIO_INEN) shown in Figure 14-18 is used to enable the input buffers on any flag pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx pin is not used in the system. By default, the input buffers are disabled.

If the PFx pin is being used as an input, the corresponding bit in the FIO_INEN register must be set. Otherwise, changes at the flag pins will not be recognized by the processor.

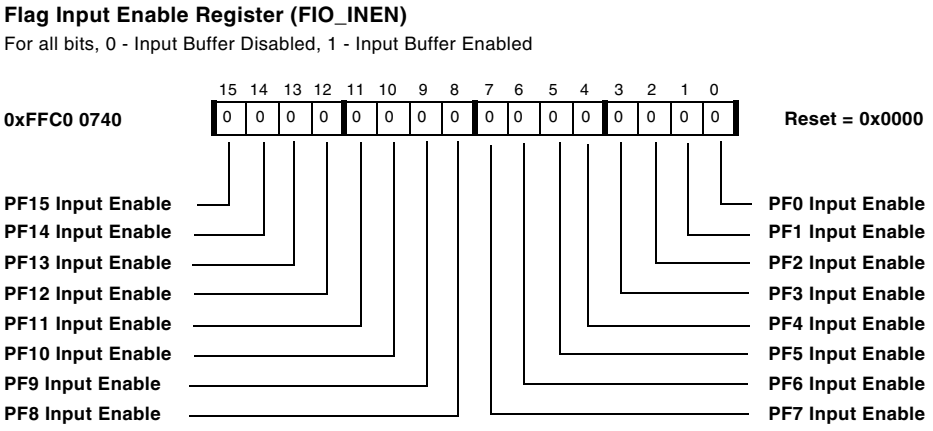


Figure 14-18. Flag Input Enable Register

Performance/Throughput

The PFx pins are synchronized to the system clock (SCLK). When configured as outputs, the programmable flags can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of PFx pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the flag is asserted and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

15 GENERAL PURPOSE INPUT/OUTPUT PORTS

The ADSP-BF539 processors have an extensive set of peripherals, all of which may not be used in an application. A GPIO (general-purpose input/output) function is multiplexed with many of the peripheral pins. GPIO functionality may be enabled on a per-pin basis in lieu of peripheral functionality. GPIO functionality differs from Programmable Flags in two ways.

- Interrupt capabilities are not associated with GPIO pins.
- GPIO enable and control are not associated with reads and writes of any peripheral registers.

GPIO pins are grouped onto Ports C through E. Each pin within a group is individually programmable. If an application's peripheral implementation does not require all of its pins, the remaining pins may be configured as GPIO. Table 15-1 shows how the peripherals are mapped to the GPIO ports.

Register nomenclature for the GPIO ports use a prefix of `GPIO_x_`, where x can be C, D, or E.

Following a system reset, all GPIO capability is disabled and a pin's functionality matches the peripheral pin's functionality. Therefore, out of reset, a peripheral pin with GPIO capability functions as if the pin were dedicated as a peripheral pin with no GPIO multiplexed functionality.

A pin can be configured to be a GPIO by writing the corresponding bit in its GPIO configuration register (GPIO_x_CNFG). Once configured as a GPIO, the pin's direction can be set using the GPIO direction registers (GPIO_x_DIR). It is permissible to pre-set the GPIO output data value before setting the GPIO direction as an output.

Table 15-1. Peripheral Multiplexing

Port	Primary Function After Reset	Alternative Function GPIO
Port C	CAN	PC 0:1
	MXVR	PC 4:9
Port D	SPI1	PD 0:4
	SPI2	PD 5:9
	UART1	PD 10:11
	UART2	PD 12:13
Port E	SPORT2	PE 0:7
	SPORT3	PE 8:15

When the GP pin is tied high, the shared pins for MXVR/GPIO take on GPIO functionality, regardless of software modifications.

There are a number of methods for controlling a GPIO's output data value through register writes. These registers include GPIO data (GPIO_x_D), data set (GPIO_x_S), data clear (GPIO_x_C), and data toggle (GPIO_x_T). These data output control methods eliminate any coherency issues normally associated with a read-modify-write sequence.

Tables Table 15-2, Table 15-3, and Table 15-4 list all of the peripheral pins which have GPIO capabilities. Each GPIO port has a complete set of registers associated with its control. The bit identified in these tables should be used when making accesses to any of the GPIO registers, as this relative bit position holds true for all the GPIO registers for that port. For example, if the application doesn't use the CAN controller, the two CAN pins (CANTX and CANRX) can be freed as GPIO by setting bits 0 and 1 in GPIO_C_CNFG. From that point forward, software can configure the pins independently and control the values driven on them by making writes to bits 0 and 1 in the associated port C GPIO registers.

Table 15-2. GPIO Port C Multiplexed Functionality Pin List

GPIO Port C Pin	Peripheral Pin Multiplexed with GPIO Pin	Associated Bit in Port C GPIO Registers
PC 0	CAN Transmit (CANTX)	Bit 0
PC 1	CAN Receive (CANRX)	Bit 1
PC 4	MXVR Receive Data (MRX)	Bit 4
PC 5	MXVR Transmit Data (MTX)	Bit 5
PC 6	MXVR Master Clock (MMCLK)	Bit 6
PC 7	MXVR Bit Clock (MBCLK)	Bit 7
PC 8	MXVR Frame Sync (MFS)	Bit 8
PC 9	MXVR Transmit FOT Enable ($\overline{\text{MTXON}}$)	Bit 9

Table 15-3. GPIO Port D Multiplexed Functionality Pin List

GPIO Port D Pin	Peripheral Pin Multiplexed with GPIO Pin	Associated Bit in Port D GPIO Registers
PD 0	SPI1 Master Out Slave In (MOSI1)	Bit 0
PD 1	SPI1 Master In Slave Out (MISO1)	Bit 1
PD 2	SPI1 Clock (SCK1)	Bit 2
PD 3	SPI1 Slave Select Input ($\overline{\text{SPI1SS}}$)	Bit 3
PD 4	SPI1 Slave Select Enable ($\overline{\text{SPI1SEL}}$)	Bit 4
PD 5	SPI2 Master Out Slave In (MOSI2)	Bit 5
PD 6	SPI2 Master In Slave Out (MISO2)	Bit 6
PD 7	SPI2 Clock (SCK2)	Bit 7
PD 8	SPI2 Slave Select Input ($\overline{\text{SPI2SS}}$)	Bit 8
PD 9	SPI2 Slave Select Enable ($\overline{\text{SPI2SEL}}$)	Bit 9
PD 10	UART1 Receive (RX1)	Bit 10
PD 11	UART1 Transmit (TX1)	Bit 11
PD 12	UART2 Receive (RX2)	Bit 12
PD 13	UART2 Transmit (TX2)	Bit 13

Table 15-4. GPIO Port E Multiplexed Functionality Pin List

GPIO Port E Pin	Peripheral Pin Multiplexed with GPIO Pin	Associated Bit in Port E GPIO Registers
PE 0	SPORT2 Receive Serial Clock (RSCLK2)	Bit 0
PE 1	SPORT2 Receive Frame Sync (RFS2)	Bit 1
PE 2	SPORT2 Receive Data Primary (DR2PRI)	Bit 2
PE 3	SPORT2 Receive Data Secondary (DR2SEC)	Bit 3
PE 4	SPORT2 Transmit Serial Clock (TSCLK2)	Bit 4
PE 5	SPORT2 Transmit Frame Sync (TFS2)	Bit 5
PE 6	SPORT2 Transmit Data Primary (DT2PRI)	Bit 6
PE 7	SPORT2 Transmit Data Secondary (DT2SEC)	Bit 7

Table 15-4. GPIO Port E Multiplexed Functionality Pin List

GPIO Port E Pin	Peripheral Pin Multiplexed with GPIO Pin	Associated Bit in Port E GPIO Registers
PE 8	SPORT3 Receive Serial Clock (RSCLK3)	Bit 8
PE 9	SPORT3 Receive Frame Sync (RFS3)	Bit 9
PE 10	SPORT3 Receive Data Primary (DR3PRI)	Bit 10
PE 11	SPORT3 Receive Data Secondary (DR3SEC)	Bit 11
PE 12	SPORT3 Transmit Serial Clock (TSCLK3)	Bit 12
PE 13	SPORT3 Transmit Frame Sync (TFS3)	Bit 13
PE 14	SPORT3 Transmit Data Primary (DT3PRI)	Bit 14
PE 15	SPORT3 Transmit Data Secondary (DT3SEC)	Bit 15

GPIO Memory-Mapped Registers (MMRs)

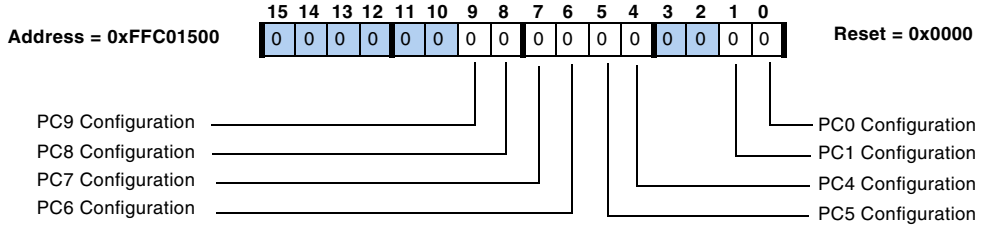
These registers are part of the system memory- mapped registers (MMRs). The addresses of the GPIO MMRs appear in Chapter B, "System MMR Assignments". Core access to the GPIO registers is through the system bus.

GPIO Configuration (GPIO_x_CNFG) Register

The GPIO configuration register (`GPIO_x_CNFG`) is a read-write register. Each bit position corresponds to a GPIO pin. The reset value of these registers is 0x0000. In this state, all GPIO capable pins are disabled and function per their respective peripheral pin definitions. A logic one written to a bit position disables the peripheral pin's definition and enables the pin as a GPIO. A logic 0 returns the pin's function to the peripheral pin's definition. A read of unused `GPIO_x_CNFG` bits always returns a value of 0, while a write of unused bits has no effect.

GPIO Port C Configuration Register (GPIO_C_CNFG)

For all bits, 0 - peripheral pin function, 1 - enable GPIO mode



PC1 and PC4 are 5V-tolerant. When configured as outputs, they behave as open-drains.

When the GP pin is pulled high to disable the MXVR, pins PC[9:4] are automatically configured as GPIO and these bits have no effect.

Figure 15-1. GPIO Port C Configuration Register

GPIO Port D Configuration Register (GPIO_D_CNFG)

For all bits, 0 - peripheral pin function, 1 - enable GPIO mode

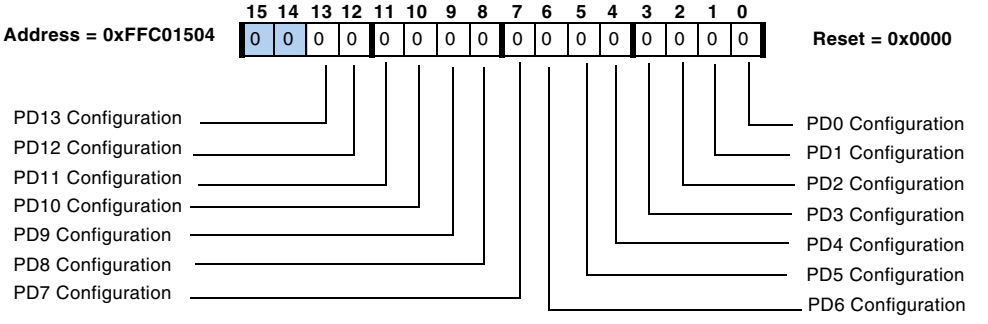


Figure 15-2. GPIO Port D Configuration Register

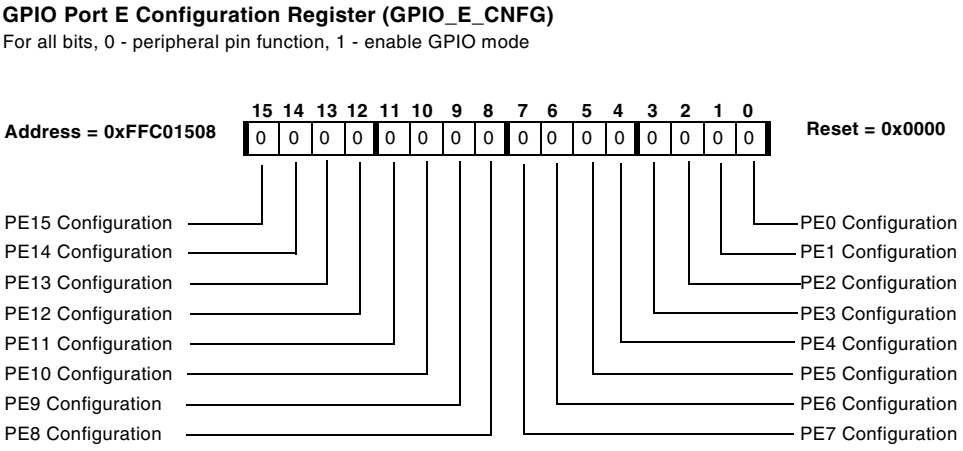
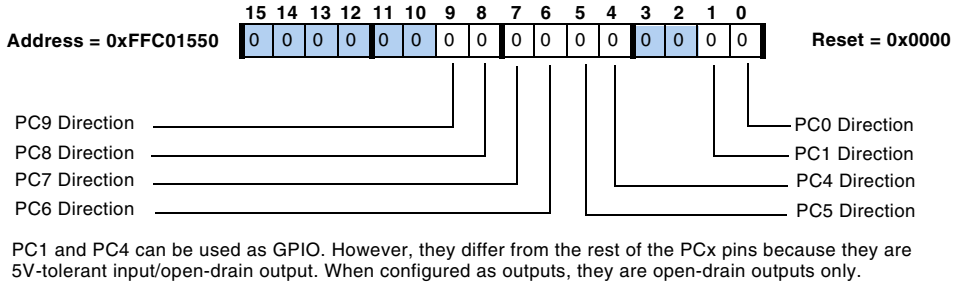


Figure 15-3. GPIO Port E Configuration Register

GPIO Direction (GPIO_x_DIR) Register

The GPIO direction register is a read-write register. Each bit position corresponds to a GPIO pin. If a pin is configured as a GPIO (`GPIO_x_CNFG`), a logic 1 enables the GPIO pin as an output, driving the state contained in the associated `GPIO_x_D` register. A logic 0 configures the GPIO pin as an input. The reset value is 0x0000, making all GPIO pins inputs by default when enabled as GPIO. A read of unused `GPIO_x_DIR` bits always returns a value of 0, while a write of unused bits has no effect.

GPIO Port C Direction Register (GPIO_C_DIR)
For all bits, 0 - input, 1 - output



PC1 and PC4 can be used as GPIO. However, they differ from the rest of the PCx pins because they are 5V-tolerant input/open-drain output. When configured as outputs, they are open-drain outputs only.

Figure 15-4. GPIO Port C Direction Register

GPIO Port D Direction Register (GPIO_D_DIR)
For all bits, 0 - input, 1 - output

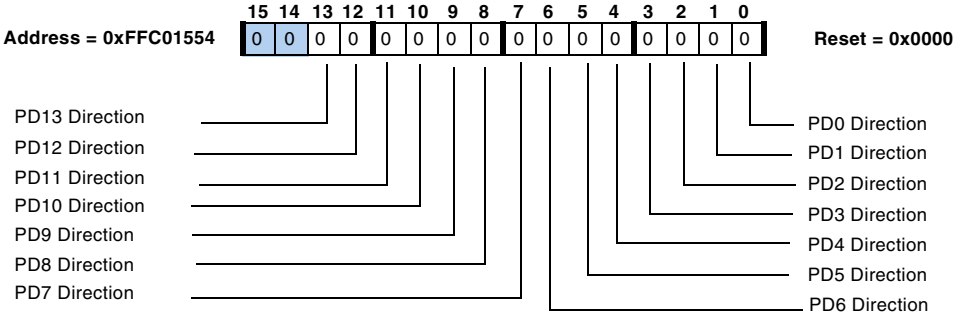


Figure 15-5. GPIO Port D Direction Register

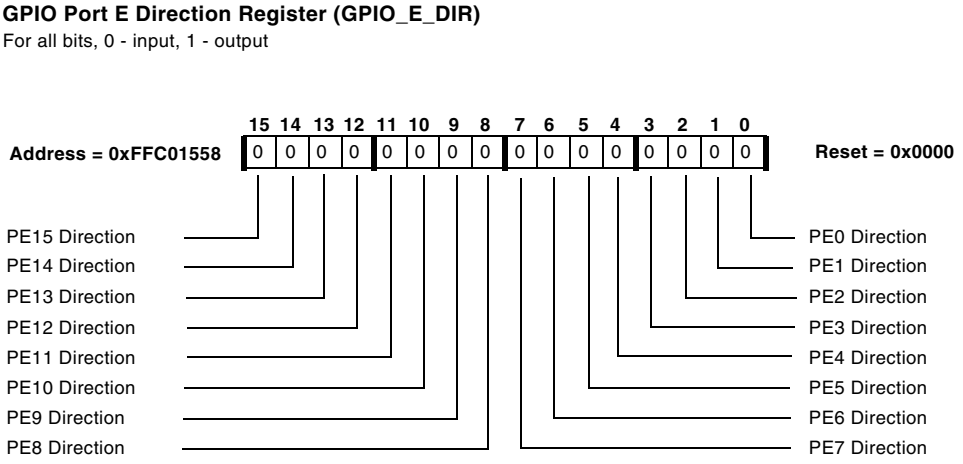


Figure 15-6. GPIO Port E Direction Register

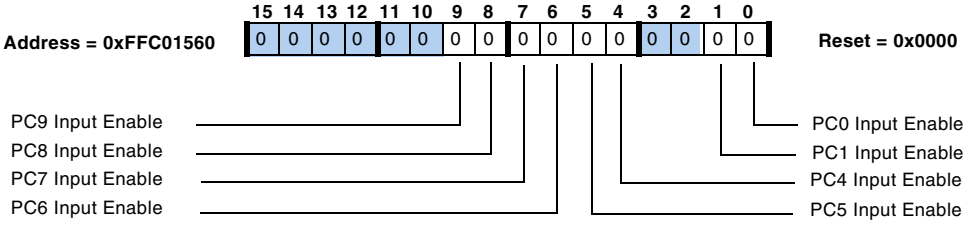
GPIO Input Enable (GPIO_x_INEN) Register

The GPIO input enable register is used to enable the input buffers on any GPIO pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pullups and pulldowns when a particular GPIO pin is not used in the system. By default, the input buffers are disabled. A read of unused `GPIO_x_INEN` bits always returns a value of 0, while a write of unused bits has no effect.

If the GPIO pin is being used as an input, the corresponding bit in the GPIO input enable register must be set.

GPIO Input Port V Enable Register (GPIO_C_INEN)

For all bits, 0 - input buffer disabled, 1 - input buffer enabled



PC1 and PC4 are 5V-tolerant input/open-drain output pins.

Figure 15-7. GPIO Port C Input Enable Register

GPIO Port D Input Enable Register (GPIO_D_INEN)

For all bits, 0 - input buffer disabled, 1 - input buffer enabled

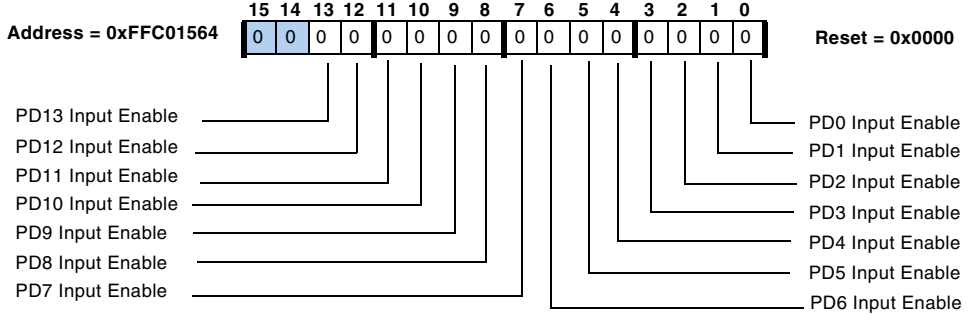


Figure 15-8. GPIO Port D Input Enable Register

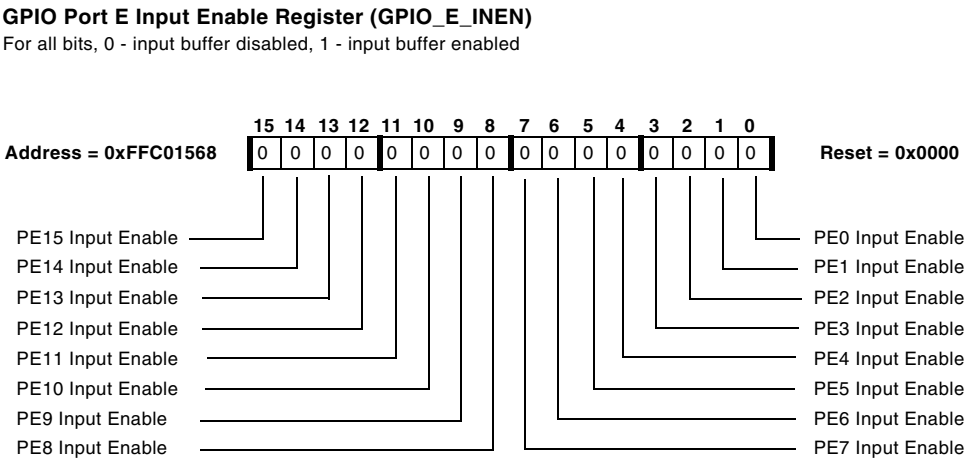


Figure 15-9. GPIO Port E Input Enable Register

GPIO Value Registers

The processor has four GPIO Value registers:

- GPIO data register (`GPIO_x_D`)
- GPIO set register (`GPIO_x_S`)
- GPIO clear register (`GPIO_x_C`)
- GPIO toggle register (`GPIO_x_T`)

These registers are used to:

- Sense the value of the GPIO pins defined as inputs
- Specify the state of GPIO pins defined as outputs

Each GPIO pin is represented by a bit in each of the four value registers.

Reading any of the GPIO data, GPIO set, GPIO clear, or GPIO toggle registers returns the value of the GPIO pins. The value returned shows the state of the GPIO pins defined as outputs and the sense of GPIO pins defined as inputs.

Reading the GPIO data, GPIO set, GPIO clear, or GPIO toggle register after reset results in 0x0000 because the pins are not enabled, even though they are reset as inputs.

For more information about the GPIO set, GPIO clear, and GPIO toggle registers, see "GPIO Clear (GPIO_x_C), GPIO Set (GPIO_x_S), and GPIO Toggle (GPIO_x_T) Registers" on page 15-14.

GPIO Data (GPIO_x_D) Register

When written, the GPIO data register (Figure 15-4 on page 15-8) directly specifies a GPIO pin's state. When read, the register returns the value of the GPIO pins. A read of unused GPIO data bits always returns a value of 0. A write of unused `GPIO_x_D` bits has no effect.

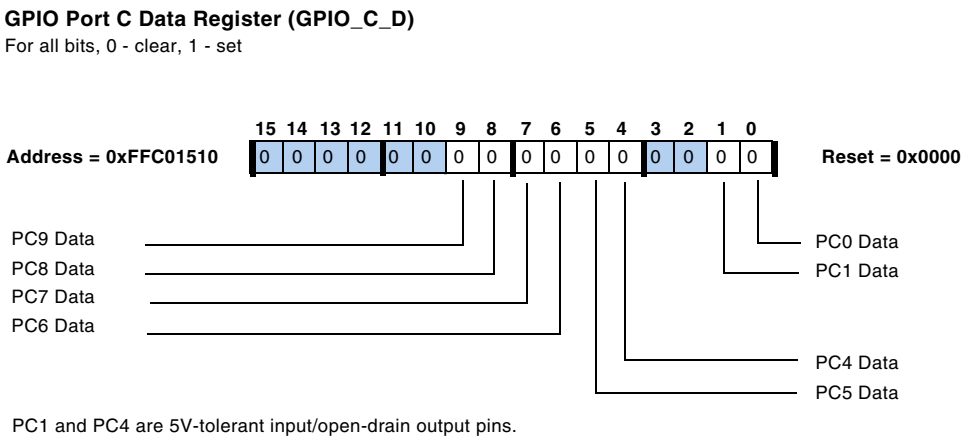


Figure 15-10. GPIO Port C Data Register

GPIO Port D Data Register (GPIO_D_D)
For all bits, 0 - clear, 1 - set

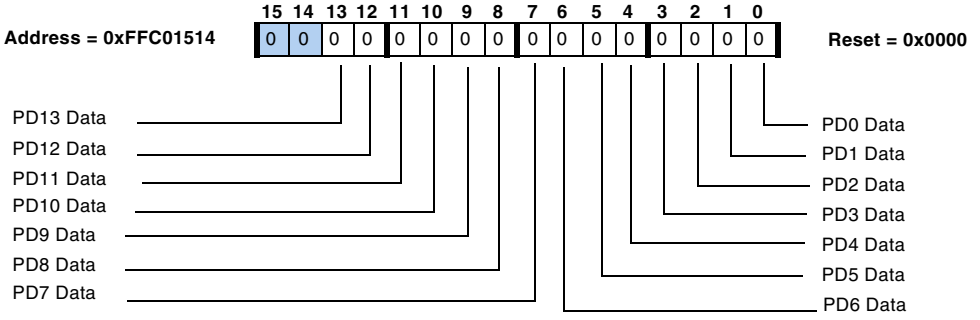


Figure 15-11. GPIO Port D Data Register

GPIO Port E Data Register (GPIO_E_D)
For all bits, 0 - clear, 1 - set

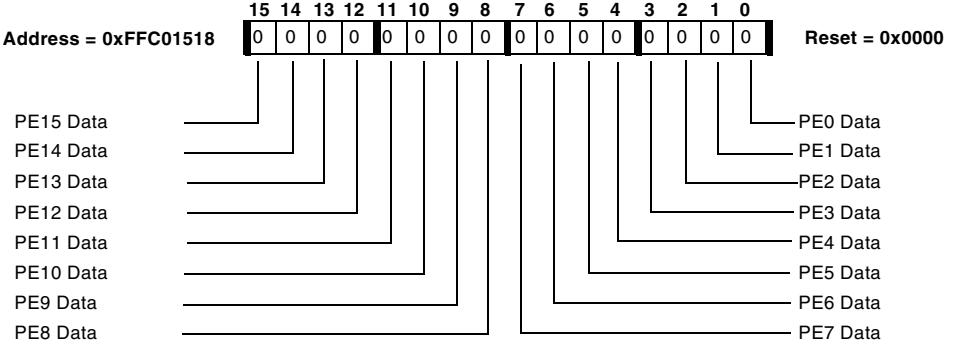


Figure 15-12. GPIO Port E Data Register

GPIO Clear (GPIO_x_C), GPIO Set (GPIO_x_S), and GPIO Toggle (GPIO_x_T) Registers

The GPIO set, GPIO clear, and GPIO toggle registers are used to set, clear, or toggle the output state associated with each output GPIO pin.

This mechanism is used to avoid the potential issues with more traditional read-modify-write mechanisms. Reading any of these registers returns the GPIO pin state. A read of unused bits in these registers always returns a value of 0. A write of unused bits has no effect.

Figure 15-13 through Figure 15-21 represent the GPIO set and GPIO clear registers, respectively.

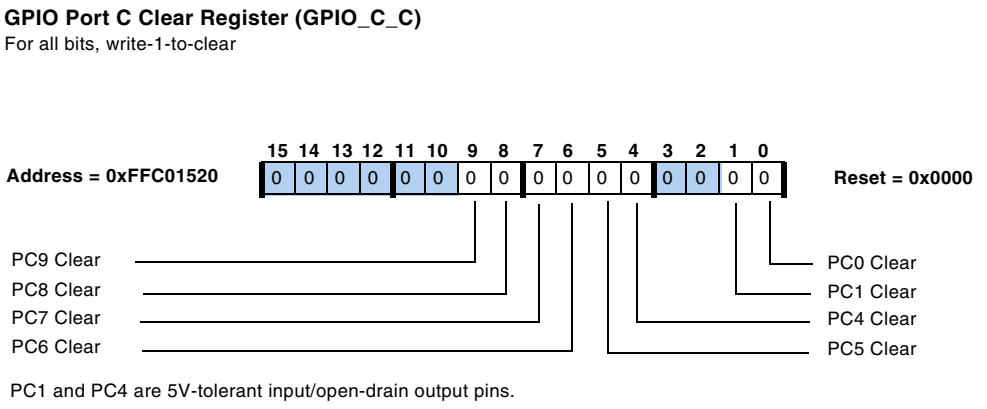


Figure 15-13. GPIO Port C Clear Register

GPIO Port D Clear Register (GPIO_D_C)
For all bits, write-1-to-clear

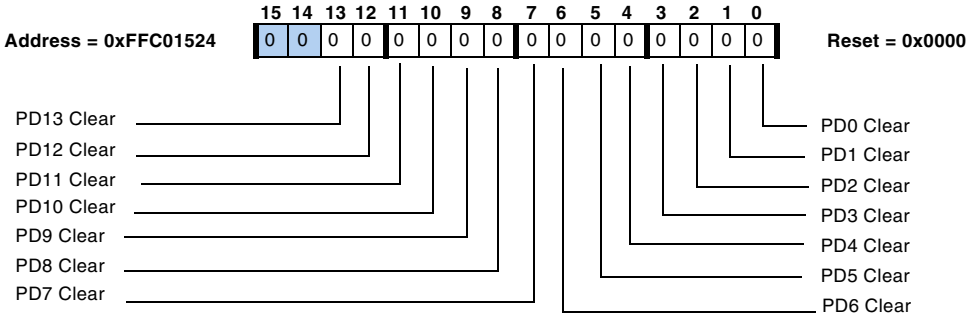


Figure 15-14. GPIO Port D Clear Register

GPIO Port E Clear Register (GPIO_E_C)
For all bits, write-1-to-clear

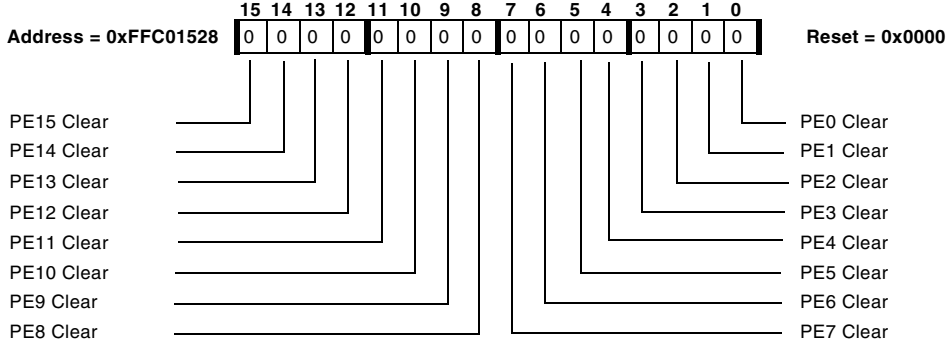


Figure 15-15. GPIO Port E Clear Register

GPIO Port C Set Register (GPIO_C_S)
For all bits, write-1-to-set

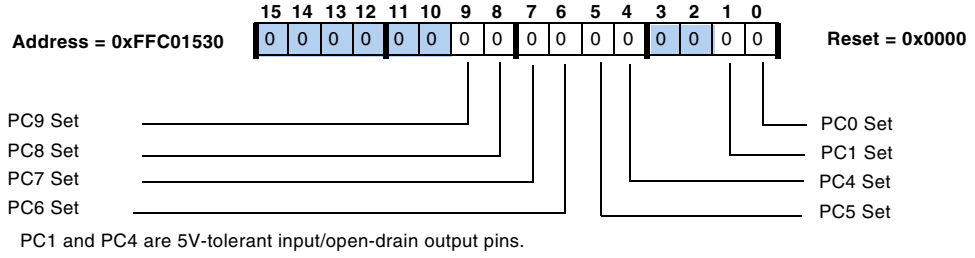


Figure 15-16. GPIO Port C Set Register

GPIO Port D Set Register (GPIO_D_S)
For all bits, 0 - write-1-to-set

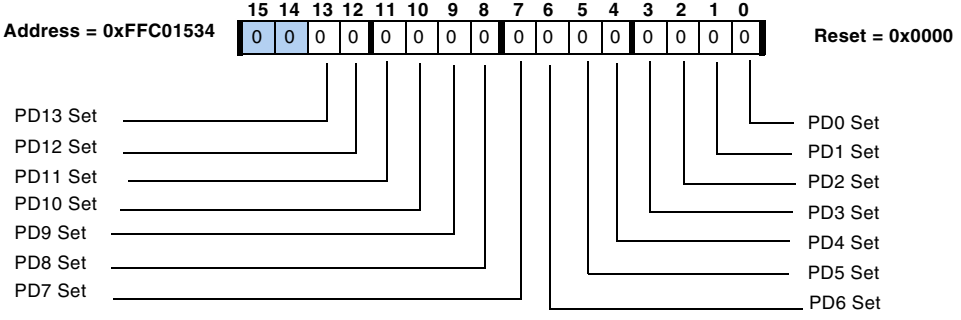


Figure 15-17. GPIO Port D Set Register

GPIO E Set Register (GPIO_E_S)
For all bits, write-1-to-set

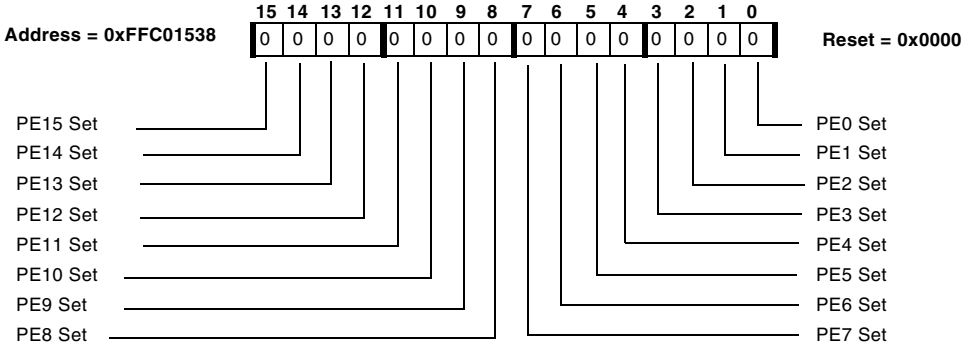
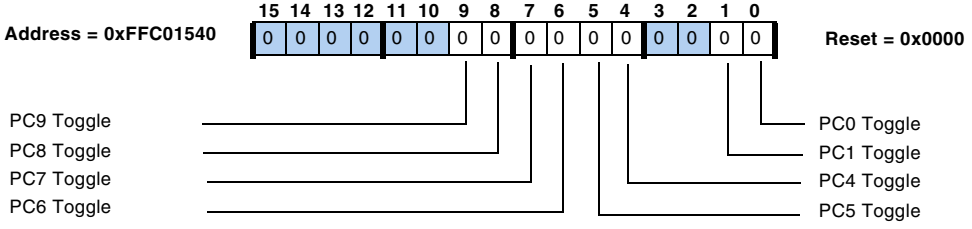


Figure 15-18. GPIO Port E Set Register

Figure 15-19, Figure 15-20, and Figure 15-21 show the GPIO toggle registers.

GPIO Port C Toggle Register (GPIO_C_T)
For all bits, write-1-to-toggle



PC1 and PC4 are 5V-tolerant input/open-drain output pins.

Figure 15-19. GPIO Port C Toggle Register

GPIO Port D Toggle Register (GPIO_D_T)
For all bits, write-1-to-toggle

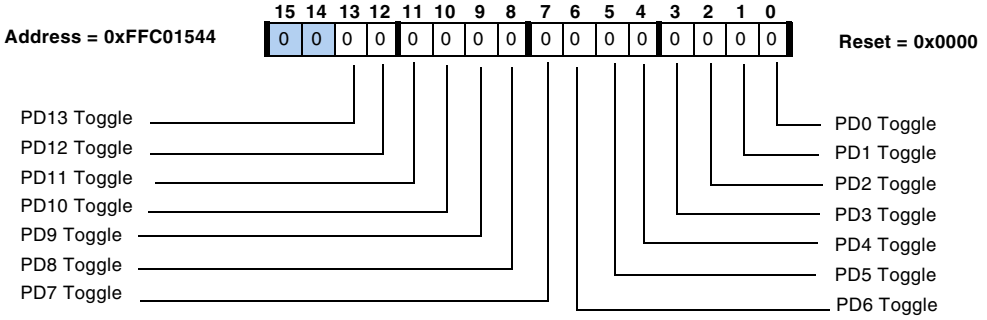


Figure 15-20. GPIO Port D Toggle Register

GPIO Port E Toggle Register (GPIO_E_T)
For all bits, write-1-to-toggle

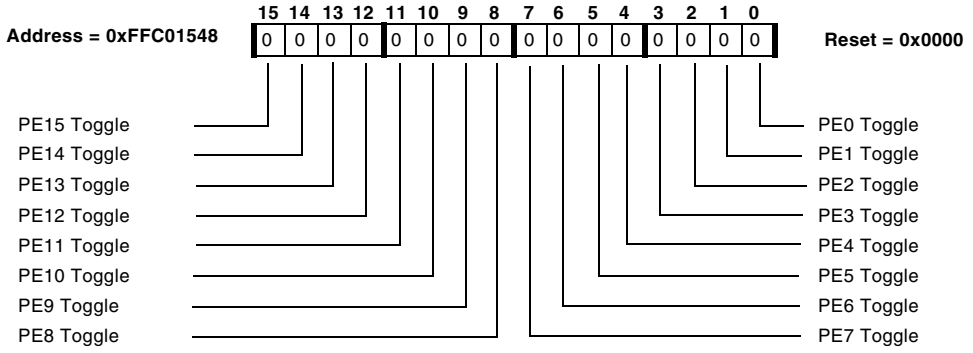


Figure 15-21. GPIO Port E Toggle Register

As an example of how these registers work, assume that PD0 is configured as an output (GPIO_D_CNFG = 0x0001 and GPIO_D_DIR = 0x0001). Writing 0x0001 to the GPIO_D_S register drives a logic 1 on the PD0 pin without affecting the state of any other GPIO pins. Writing 0x0001 to the GPIO_D_C register drives a logic 0 on the PD0 pin without affecting the state of any other GPIO pins. Writing a 0x0001 to the GPIO_D_T register changes the pin state on PD0 from logic zero to logic one or from logic one to logic zero, depending upon the existing pin state, without affecting the state of any other GPIO pins.

Writing a 0 to one of these registers has no effect on the value of the PD0 pin, and is therefore ignored.

Reading the GPIO set or GPIO clear register returns:

- 0 for GPIO pins defined as outputs and driven low
- 1 for pins (including PD0 in the example above) defined as outputs and driven high
- The present sense of GPIO pins defined as inputs

Performance/Throughput

The GPIO pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition once every system clock cycle.

16 TIMERS

The processor features three identical 32-bit general-purpose timers, a core timer, and a watchdog timer.

The general-purpose timers can be individually configured in any of three modes:

- Pulse width modulation (PWM_OUT) mode
- Pulse width count and capture (WDTH_CAP) mode
- External event (EXT_CLK) mode

The core timer is available to generate periodic interrupts for a variety of system timing functions.

The watchdog timer can be used to implement a software watchdog function. A software watchdog can improve system availability by generating an event to the Blackfin processor core if the timer expires before being updated by software.

General-purpose Timers

Each general-purpose timer has one dedicated bidirectional chip pin, TMRx. This pin functions as an output pin in the PWM_OUT mode and as an input pin in the WDTH_CAP and EXT_CLK modes. To provide these functions, each timer has four registers. For range and precision, the timer counter (TIMERx_COUNTER), timer period (TIMERx_PERIOD), and timer pulse width (TIMERx_WIDTH) registers are 32 bits wide. See Figure 16-1.

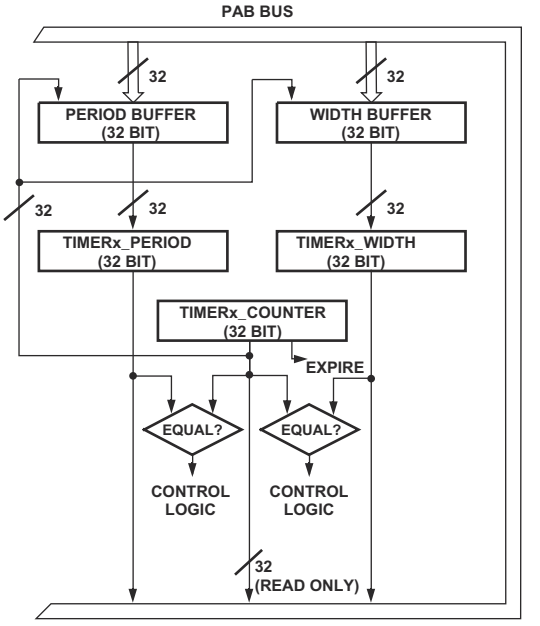


Figure 16-1. Timer Block Diagram

The registers for each general-purpose timer are:

- Timer configuration (TIMERx_CONFIG) registers
- Timer counter (TIMERx_COUNTER) registers
- Timer period (TIMERx_PERIOD) registers
- Timer pulse Width (TIMERx_WIDTH) registers

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is ($(2^{32}$-1) / 133 MHz) = 32.2 seconds.

The timer enable (TIMER_ENABLE) register can be used to enable all three timers simultaneously. The register contains three “write-1-to-set” control bits, one for each timer. Correspondingly, the timer disable (TIMER_DISABLE) register contains three “write-1-to-clear” control bits to allow simultaneous or independent disabling of the three timers. Either the timer enable or the timer disable register can be read back to check the enable status of the timers. A 1 indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMENx bit is set.

The timer status (TIMER_STATUS) register contains an interrupt latch bit (TIMILx) and an overflow/error indicator bit (TOVF_ERRx) for each timer. These sticky bits are set by the timer hardware and may be polled by software. They need to be cleared by software explicitly, by writing a 1 to the bit.

To enable a timer’s interrupts, set the IRQ_ENA bit in the timer’s configuration (TIMERx_CONFIG) register and unmask the timer’s interrupt by setting the corresponding bits of the IMASK and SIC_IMASKx registers. With the IRQ_ENA bit cleared, the timer does not set its timer interrupt latch (TIMILx) bits. To poll the TIMILx bits without permitting a timer interrupt, programs can set the IRQ_ENA bit while leaving the timer’s interrupt masked.

With interrupts enabled, make sure that the interrupt service routine (ISR) clears the TIMILx latch before the RTI instruction, to ensure that the interrupt is not reissued. To make sure that no timer event is missed, the latch should be reset at the very beginning of the interrupt routine when in external clock (EXT_CLK) mode. To enable timer interrupts, set the IRQ_ENA bit in the proper timer configuration (TIMERx_CONFIG) register.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of three identical timer units.

Each timer provides four registers:

- TIMERx_CONFIG[15:0] – Timer configuration register
- TIMERx_WIDTH[31:0] – Timer pulse width register
- TIMERx_PERIOD[31:0] – Timer period register
- TIMERx_COUNTER[31:0] – Timer counter register

Three registers are shared between the three timers:

- TIMER_ENABLE[15:0] – Timer enable register
- TIMER_DISABLE[15:0] – Timer disable register
- TIMER_STATUS[15:0] – Timer status register

The size of accesses is enforced. A 32-bit access to a timer configuration register or a 16-bit access to a timer pulse width, timer period, or timer counter register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the timer enable, timer disable, and timer status registers. On a 32-bit read, the upper word returns all 0s.

TIMER_ENABLE Register

The timer enable register (TIMER_ENABLE) allows all three timers to be enabled simultaneously in order to make them run completely synchronously. For each timer there is a single W1S control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The three bits can be set individually or in any combination. A read of the Timer Enable register shows the status of the enable for the corresponding timer. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

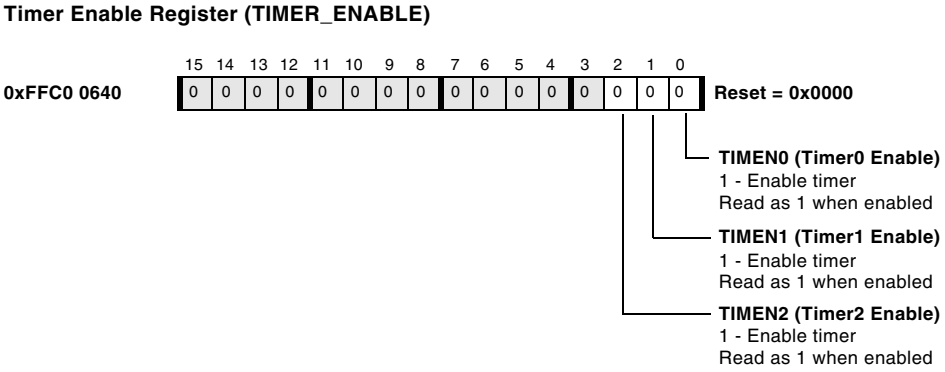


Figure 16-2. Timer Enable Register

TIMER_DISABLE Register

The timer disable register (TIMER_DISABLE) allows all three timers to be disabled simultaneously. For each timer there is a single W1C control bit. Writing a 1 disables the corresponding timer; writing a 0 has no effect. The three bits can be cleared individually or in any combination. A read of the timer disable register returns a value identical to a read of the Timer Enable register. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

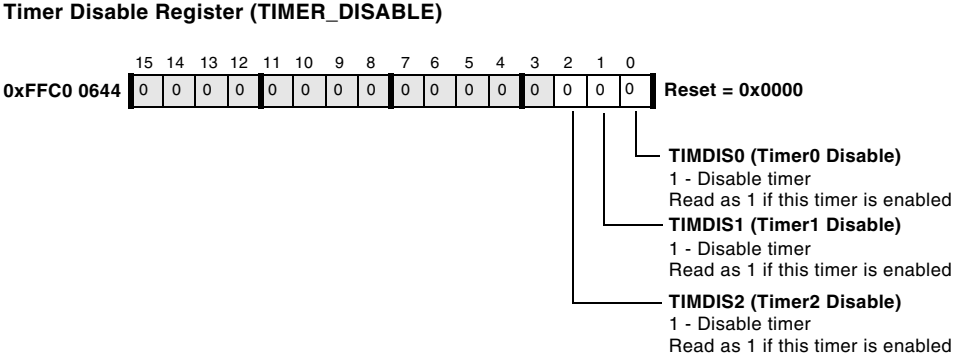


Figure 16-3. Timer Disable Register

In PWM_OUT mode, a write of a 1 to TIMER_DISABLE does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a 1 to the corresponding bit in TIMER_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 16-20.

In WDTH_CAP and EXT_CLK modes, a write of a 1 to TIMER_DISABLE stops the corresponding timer immediately.

TIMER_STATUS Register

The Timer status register (TIMER_STATUS) indicates the status of all three timers and is used to check the status of all three timers with a single read. status bits are sticky and W1C. The TRUNx bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a status register read access, all reserved or unused bits return a 0.

Each Timer generates a unique interrupt request signal, which is gated by the corresponding IRQ_ENA bit in the TIMERx_CONFIG register. The shared Timer status register (TIMER_STATUS) latches these interrupts so the user can determine the interrupt source without reference to the unique interrupt signal (for example, in the case where all three timers have been assigned to the same interrupt priority). Interrupt bits are sticky and must be cleared by the interrupt service routine (ISR) to assure that the interrupt is not reissued.

The TIMILx bits work along with the IRQ_ENA bit of the Timer Configuration register to indicate interrupt requests. If an interrupt condition or error occurs and IRQ_ENA is set, then the TIMILx bit is set and the interrupt to the core is asserted. This interrupt may be masked by the system interrupt controllers. If an interrupt condition or error occurs and IRQ_ENA is

cleared, then the TIMILx bit is not set and the interrupt is not asserted. If TIMILx is already set and IRQ_ENA is written to 0, TIMILx stays set and the interrupt stays asserted. See Figure 16-24 on page -41.

The read value of the TRUNx bits reflects the timer slave enable status in all modes—TRUNx set indicates running and TRUNx cleared indicates stopped. While reading the TIMENx or TIMDISx bits in the TIMER_ENABLE and TIMER_DISABLE registers will reflect whether a timer is enabled, the TRUNx bits indicate whether the timer is actually running. In WDTH_CAP and EXT_CLK modes, reads from TIMENx and TRUNx always return the same value.

A W1C operation to the TIMER_DISABLE register disables the corresponding timer in all modes. In PWM_OUT mode, a disabled timer continues running until the ongoing period (PERIOD_CNT = 1) or pulse (PERIOD_CNT = 0) completes. During this final period the TIMENx bit returns 0, but the TRUNx bit still reads as a 1. See Figure 16-10 on page -15. In this state only, TRUNx becomes a W1C bit. During this final period with the timer disabled, writing a 1 to TRUNx clears TRUNx and stops the timer immediately without waiting for the timer counter to reach the end of its current cycle.

Writing the TRUNx bits has no effect in other modes or when a timer has not been enabled. Writing the TRUNx bits to 1 in PWM_OUT mode has no effect on a timer that has not first been disabled.

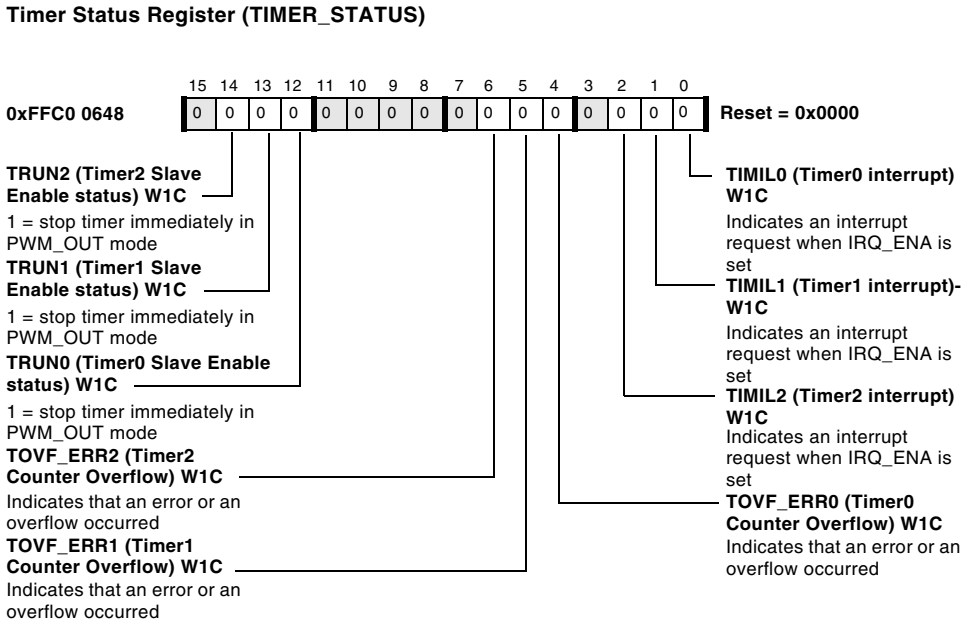


Figure 16-4. Timer status Register

TIMERx_CONFIG Registers

The operating mode for each timer is specified by its Timer Configuration register (TIMERx_CONFIG). The TIMERx_CONFIG register may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUNx bit in TIMER_STATUS before attempting to reprogram TIMERx_CONFIG. The TIMERx_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled. Each time TOVF_ERRx is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 16-1 on page -43. The TIMERx_CONFIG register also controls the behavior of the TMRx pin, which becomes an output in PWM_OUT mode (TMODE = b#01) when the OUT_DIS bit is cleared.

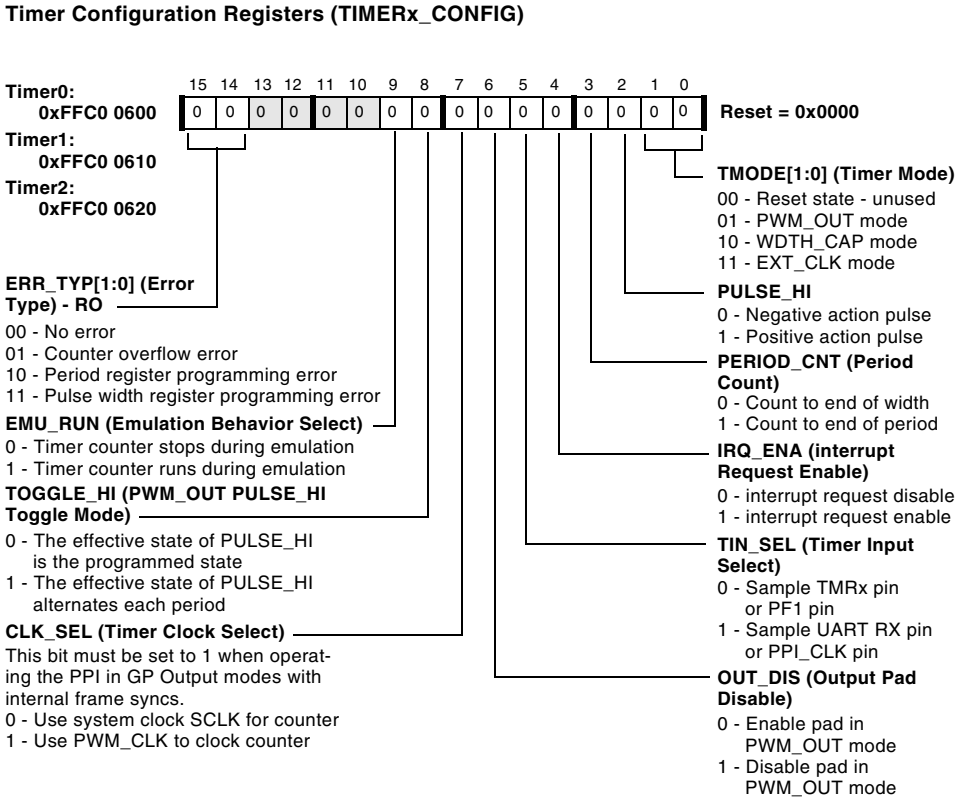


Figure 16-5. Timer Configuration Registers

TIMERx_COUNTER Registers

These read-only registers retain their state when disabled. When enabled, the Timer Counter register (TIMERx_COUNTER) is reinitialized by hardware based on configuration and mode. The Timer Counter register may be read at any time (whether the timer is running or stopped), and it returns a coherent 32-bit value. Depending on the operation mode, the incrementing counter can be clocked by four different sources: SCLK, the TMRx pin, the Programmable Flag pin PF1, or the parallel port clock PPI_CLK.

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMERx_COUNTER also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMRx pin waveform is "stretched"; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on TMRx may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMERx_PERIOD and TIMERx_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMERx_CONFIG to enable this behavior.

Timer Counter Registers (TIMERx_COUNTER)

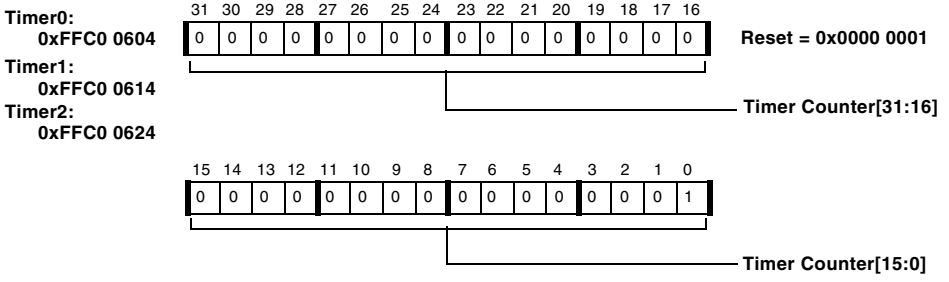


Figure 16-6. Timer Counter Registers

TIMERx_PERIOD and TIMERx_WIDTH Registers

When a timer is enabled and running, and the software writes new values to the Timer Period register and the Timer Pulse Width register, the writes are buffered and do not update the registers until the end of the current period (when the Timer Counter register equals the Timer Period register).

Usage of the Timer Period register (TIMERx_PERIOD) and the Timer Pulse Width register (TIMERx_WIDTH) varies depending on the mode of the timer:

- In Pulse Width Modulation mode (PWM_OUT), both the Timer Period and Timer Pulse Width register values can be updated "on-the-fly" since the Timer Period and Timer Pulse Width (duty cycle) register values change simultaneously.
- In Pulse Width and Period Capture mode (WDTH_CAP), the Timer Period and Timer Pulse Width buffer values are captured at the appropriate time. The Timer Period and Timer Pulse Width registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In External Event Capture mode (EXT_CLK), the Timer Period register is writable and can be updated "on-the-fly." The Timer Pulse Width register is not used.

If new values are not written to the Timer Period register or the Timer Pulse Width register, the value from the previous period is reused. Writes to the 32-bit Timer Period register and Timer Pulse Width register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the Timer Period registers or Timer Pulse Width registers are always stored in the buffer registers. Reads from the Timer Period or Timer Pulse Width registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the Timer Period and Timer Pulse Width registers are updated from their respective buffers at the end of the current period. See Figure 16-1 on page -2.

When the timer is disabled, writes to the buffer registers are immediately copied through to the Timer Period or Timer Pulse Width register so that they will be ready for use in the first timer period. For example, to change the values for the Timer Period and/or Timer Pulse Width registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In `PWM_OUT` mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the Timer Period register and the Timer Pulse Width register. The next period may use one old value and one new value. In order to prevent Pulse Width >= Period errors, write the Timer Pulse Width register before the Timer Period register when decreasing the values, and write the Timer Period register before the Timer Pulse Width register when increasing the value.

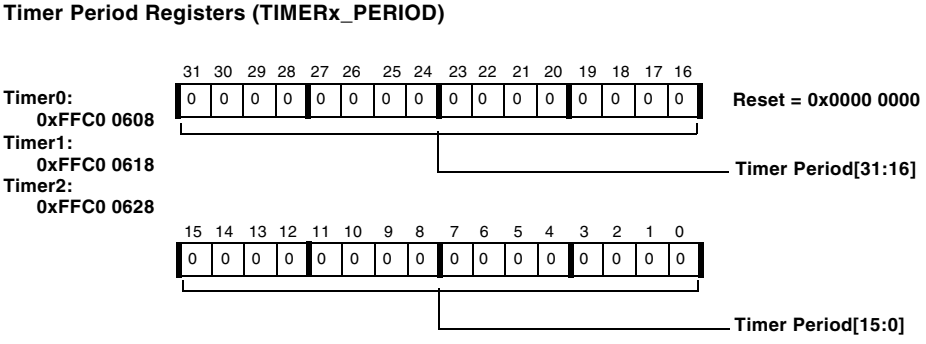


Figure 16-7. Timer Period Registers

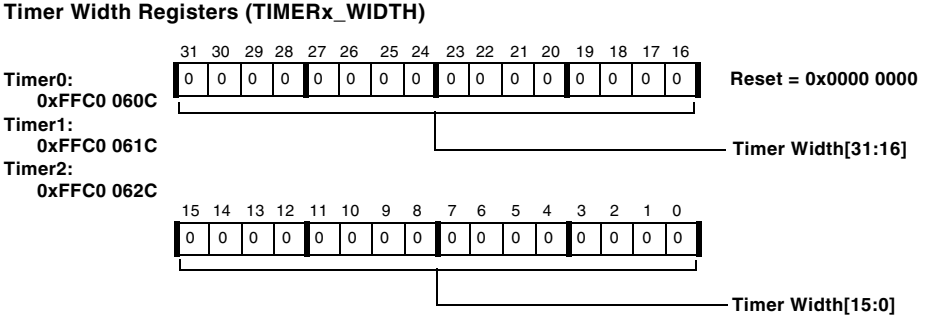


Figure 16-8. Timer Width Registers

Using the Timer

To enable an individual timer, set that timer's TIMEN bit in the TIMER_ENABLE register. To disable an individual timer, set that timer's TIMDIS bit in the TIMER_DISABLE register. To enable all three timers in parallel, set all three TIMEN bits in the TIMER_ENABLE register.

Before enabling a timer, always program the corresponding Timer Configuration (TIMERx_CONFIG) register. This register defines the timer operating mode, the polarity of the TMRx pin, and the timer interrupt behavior. Do not alter the operating mode while the timer is running.

Examples of timer enable and disable timing appear in Figure 16-9, Figure 16-10, and Figure 16-11.

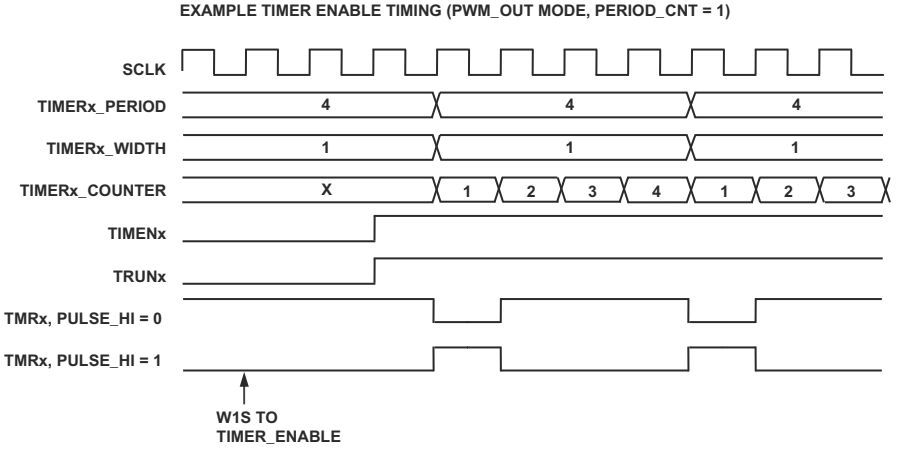


Figure 16-9. Timer Enable Timing

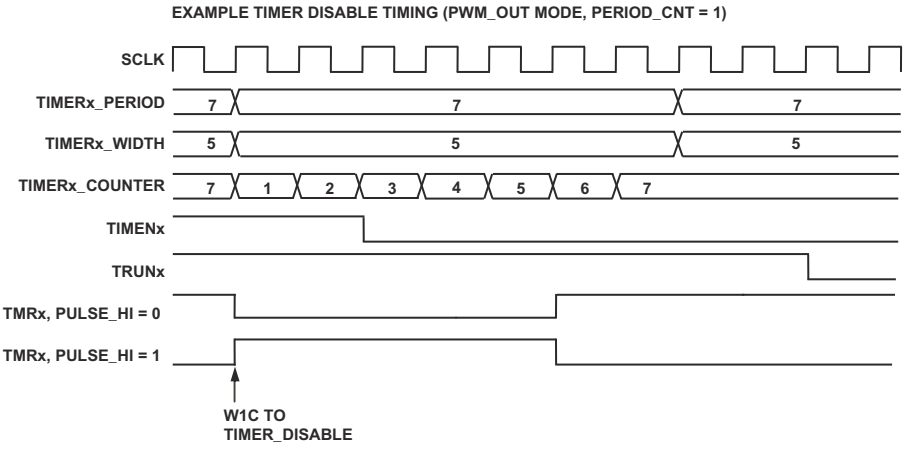


Figure 16-10. Timer Disable Timing

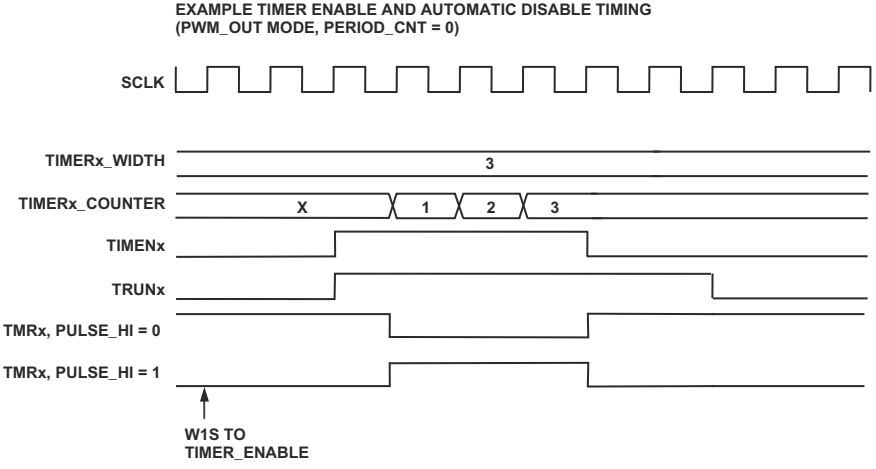


Figure 16-11. Timer Enable and Automatic Disable Timing

When timers are disabled, the Timer Counter registers retain their state; when a timer is re-enabled, the Timer Counter is reinitialized based on the operating mode. The Timer Counter registers are read-only. Software cannot overwrite or preset the Timer Counter value directly.

Pulse Width Modulation (PWM_OUT) Mode

Setting the TMODE field to b#01 in the Timer Configuration (TIMERx_CONFIG) register enables PWM_OUT mode. In PWM_OUT mode, the timer TMRx pin is an output. The output can be disabled by setting the OUT_DIS bit in the Timer Configuration register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

Once a timer has been enabled, the Timer Counter register is loaded with a starting value. If CLK_SEL = 0, the Timer Counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the Timer Period register. For either setting of CLK_SEL, when the Timer Counter equals the Timer Period, the Timer Counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMERx_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMERx_PERIOD and TIMERx_WIDTH registers and generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of PERIOD_CNT = 0 counts to the end of the Width; a setting of PERIOD_CNT = 1 counts to the end of the Period.

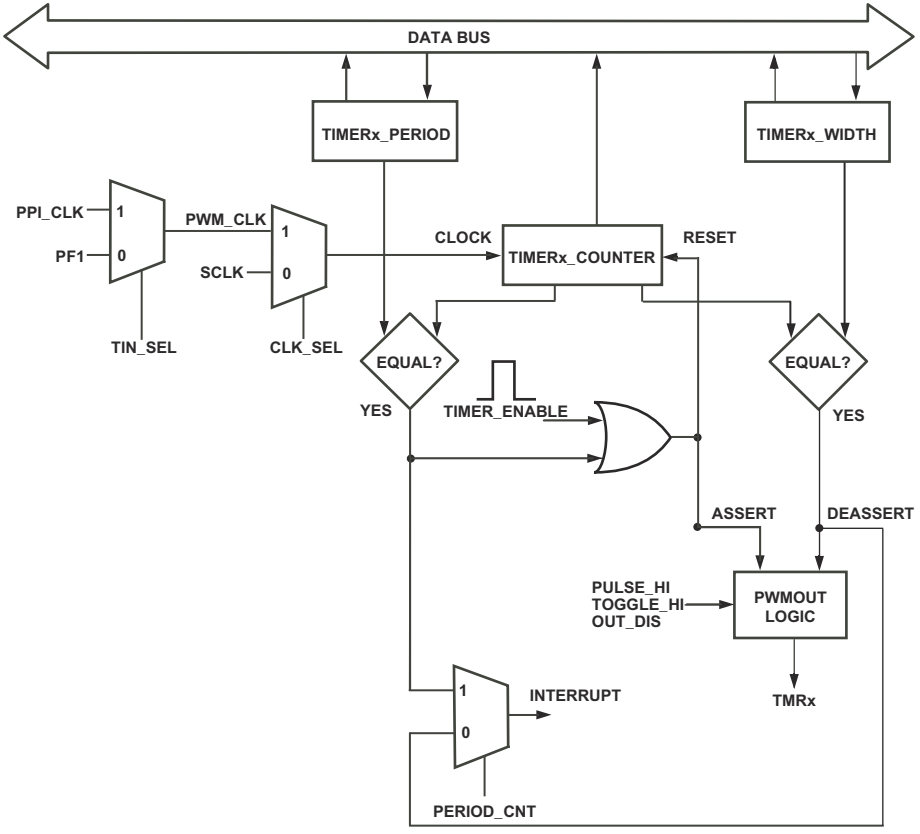


Figure 16-12. Timer Flow Diagram, PWM_OUT Mode

The TIMERx_PERIOD and TIMERx_WIDTH registers are read-only in some operation modes. Be sure to set the TMODE field in the TIMERx_CONFIG register to b#01 before writing to these registers.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the Timer Configuration register. The TMRx pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMRx pin. This mode can also be used to implement a precise delay. The pulse width is defined by the Timer Pulse Width register, and the Timer Period register is not used.

At the end of the pulse, the Timer interrupt latch bit TIMILx gets set, and the timer is stopped automatically. If the PULSE_HI bit is set, an active high pulse is generated on the TMRx pin. If PULSE_HI is not set, the pulse is active low.

Pulse Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle. This mode also generates periodic interrupts for real-time signal processing.

The 32-bit Timer Period (TIMERx_PERIOD) and Timer Pulse Width (TIMERx_WIDTH) registers are programmed with the values of the timer count period and pulse width modulated output pulse width.

When the timer is enabled in this mode, the TMRx pin is pulled to a deasserted state each time the Timer Counter equals the value of the Timer Pulse Width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMRx pin, the PULSE_HI bit in the corresponding TIMERx_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMRx pin is driven to the deasserted level.

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine (ISR) must clear the interrupt Latch bit (TIMILx) and might alter period and/or width values. In pulse width modulation (PWM) applications, the software needs to update period and pulse width values while the timer is running. When software updates either period or pulse width registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. New Timer Period and Timer Pulse Width register values are written while the old values are being used. The new values are loaded in to be used when the Timer Counter value equals the current Timer Period value. Reads from Timer Period and Timer Pulse Width registers return the old values until the period expires.

The TOVF_ERRx status bit signifies an error condition in PWM_OUT mode. The TOVF_ERRx bit is set if TIMERx_PERIOD = 0 or TIMERx_PERIOD = 1 at startup, or when the Timer Counter register rolls over. It is also set when the Timer Counter register rolls over if the Timer Pulse Width register is greater than or equal to the Timer Period register. The ERR_TYP bits are set when the TOVF_ERRx bit is set.

To generate the maximum frequency on the TMRx output pin, set the period value to 2 and the pulse width to 1. This makes TMRx toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (Period – 1), inclusive. When PERIOD_CNT = 0, the pulse width may be programmed to any value from 1 to (2^{32} – 1), inclusive.

Although the hardware reports an error if the TIMERx_WIDTH value equals the TIMERx_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore

the TOVL_ERRx flags. Pulse width values greater than the period value are not recommended. Similarly, TIMERx_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a "stop is pending" condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMRx pin. The processor can determine when the timer stops running by polling for the corresponding TRUNx bit in the TIMER_STATUS register to read 0 or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMERx_CONFIG cannot be written to a new value) until after the timer stops and TRUNx reads 0.

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and TIMER_DISABLE) is cleared, and the corresponding TRUNx bit is cleared. See Figure 16-11 on page -15. To generate multiple pulses, write a 1 to TIMER_ENABLE, wait for the timer to stop, then write another 1 to TIMER_ENABLE.

If necessary, the processor can force a timer in PWM_OUT mode to stop immediately. Do this by first writing a 1 to the corresponding bit in TIMER_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMRx pin.

In PWM_OUT continuous pulse mode (PERIOD_CNT = 1), each timer samples its TIMENx bit at the end of each period. It stops cleanly at the end of the first period when TIMENx is low. This implies (barring any W1C to TRUNx) that a timer that is disabled and then re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself. The timer will always stop at the end of the first pulse when PERIOD_CNT = 0.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the Timer Configuration (TIMERx_CONFIG) register is set, then the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the PF1 pin, but may be taken from the PPI_CLK pin when the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMERx_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMRx pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMRx pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The PF1 pin can only clock the timer when PF1 functions as an input pin. When any timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, then the PF1 bit in the FIO_DIR register is ignored and PF1 is forced to be an input.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMERx_WIDTH register). When two timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 16-13.

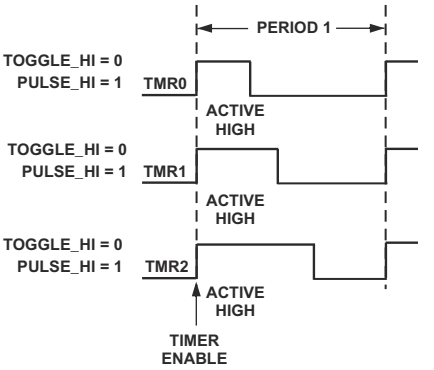


Figure 16-13. Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a fully arbitrary rectangular waveform. The effective waveform is still active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of TOGGLE_HI has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when Count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 16-14 shows an example with all three timers running with the same period settings. When software does not alter the PWM settings at runtime, the duty cycle is 50%. The values of the TIMERx_WIDTH registers control the phase between the signals.

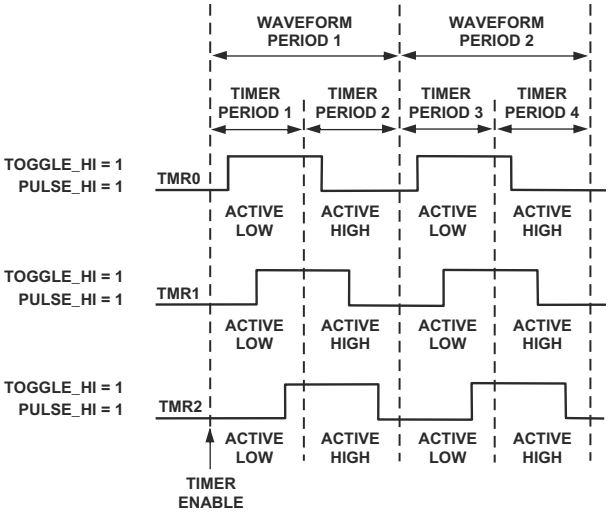


Figure 16-14. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (See Figure 16-15).

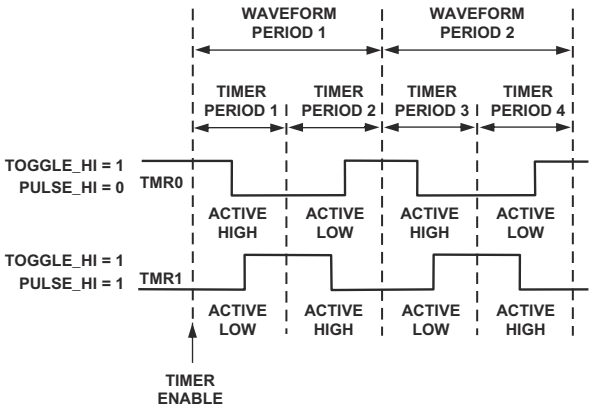


Figure 16-15. Two Timers With Non-overlapping Clocks

When `TOGGLE_HI = 0`, software updates the Timer Period and Timer Pulse Width registers once per waveform period. When `TOGGLE_HI = 1`, software updates the Timer Period and Timer Pulse Width registers twice per waveform period with values that are half as large. In odd-numbered periods, write (`Period - Width`) instead of `Width` to the Timer Pulse Width register in order to obtain center-aligned pulses.

For example, if the pseudo-code when `TOGGLE_HI = 0` is:

```
int period, width ;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;
```

```
     write(TIMERx_PERIOD, period) ;
     write(TIMERx_WIDTH, width) ;
}
```

Then when `TOGGLE_HI = 1`, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;)  {
     period = generate_period(...) ;
     width = generate_width(...) ;

     per1 = period/2 ;
     wid1 = width/2 ;

     per2 = period/2 ;
     wid2 = width/2 ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per1) ;
     write(TIMERx_WIDTH, per1 - wid1) ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per2) ;
     write(TIMERx_WIDTH, wid2) ;
}
```

As shown in this example, the pulses produced do not need to be symmetric (`wid1` does not need to equal `wid2`). The period can be offset to adjust the phase of the pulses produced (`per1` does not need to equal `per2`).

The Timer Slave Enable bit (TRUNx bit in the TIMER_STATUS register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLE is written to 1, the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when TOGGLE_HI = 0, errors are reported if:

TIMERx_WIDTH >= TIMERx_PERIOD, TIMERx_PERIOD = 0, or
TIMERx_PERIOD = 1

Pulse Width Count and Capture (WDTH_CAP) Mode

In WDTH_CAP mode, the TMRx pin is an input pin. The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the TIMERx_CONFIG (Timer Configuration register) enables this mode.

When enabled in this mode, the timer resets the count in the TIMERx_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMRx pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMERx_COUNTER register into the width buffer register. At the next leading edge, the timer transfers the current 32-bit value of the TIMERx_COUNTER register into the period buffer register. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

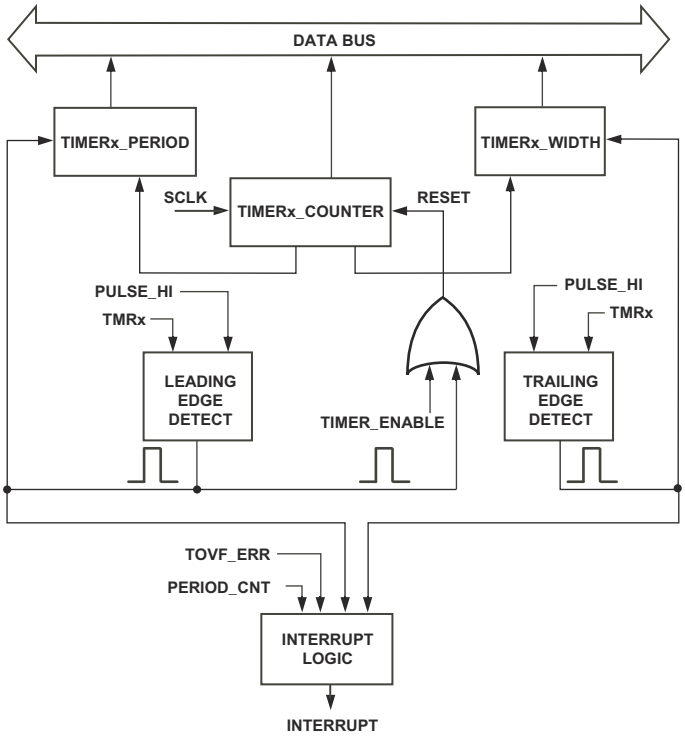


Figure 16-16. Timer Flow Diagram, WDTH_CAP Mode

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the TMRx pin, the PULSE_HI bit in the TIMERx_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the Timer Counter register is captured to the Timer Pulse Width buffer register on the rising edge, and the Timer Period is captured on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the Timer Counter register is captured to the Timer Pulse Width buffer register on the falling edge, and the Timer Period is captured on the next rising edge.

In WDTH_CAP mode, these three events always occur at the same time as one unit:

1. The TIMERx_PERIOD register is updated from the period buffer register.
2. The TIMERx_WIDTH register is updated from the width buffer register.
3. The Timer interrupt latch bit (TIMILx) gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMERx_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these three events are called a measurement report. The Timer Counter Overflow error latch bit (TOVF_ERRx) does not get set at a measurement report. A measurement report occurs at most once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMERx_PERIOD and TIMERx_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer register captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer register captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (See Figure 16-17), then the TIMERx_PERIOD and TIMERx_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (See Figure 16-18), then the TIMERx_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMERx_PERIOD register reports the pulse period measured at the end of the previous period.

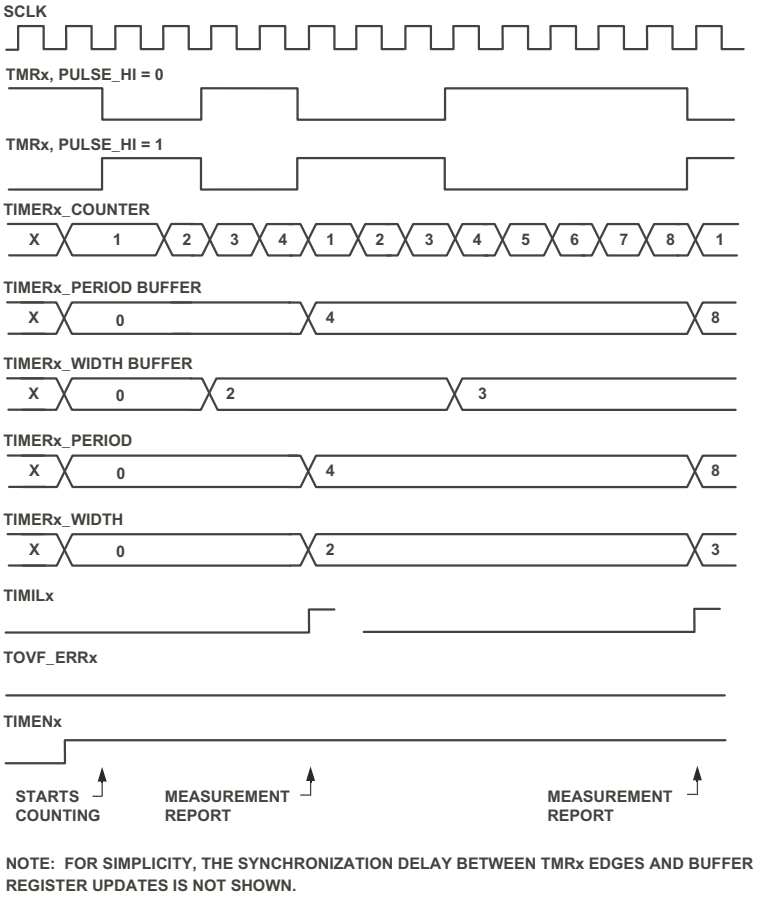
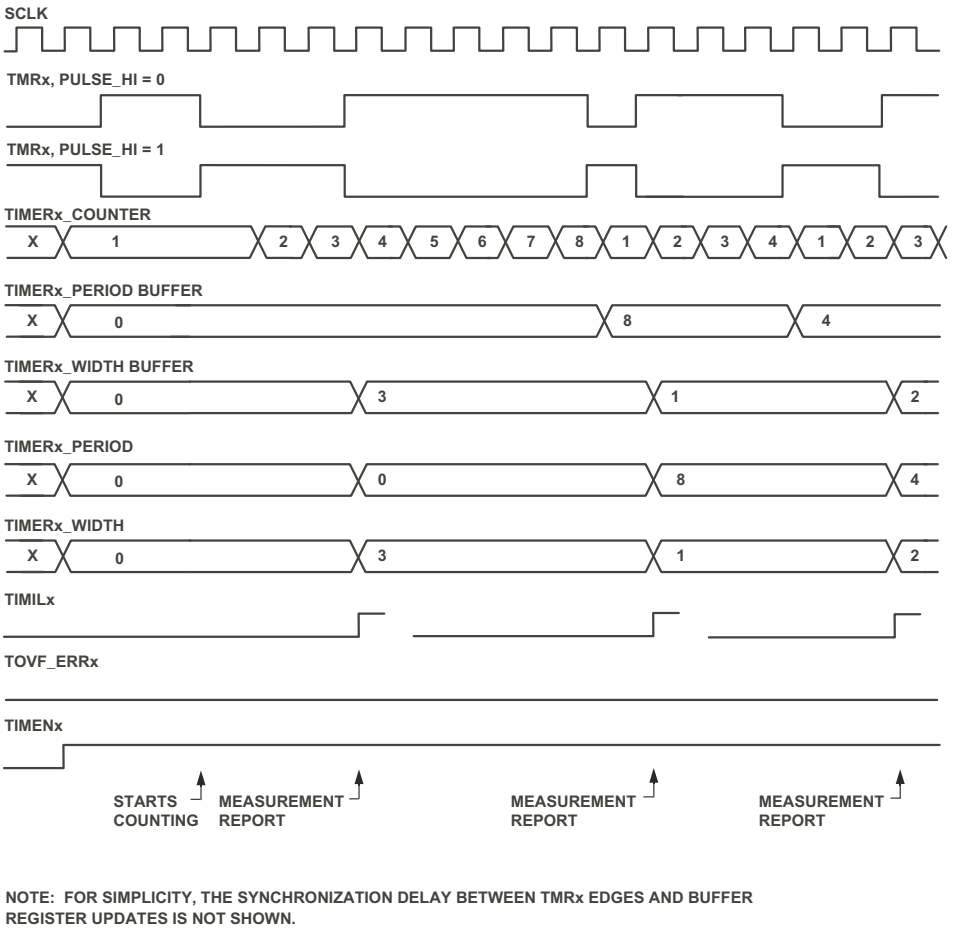


Figure 16-17. Example of Period Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 1)



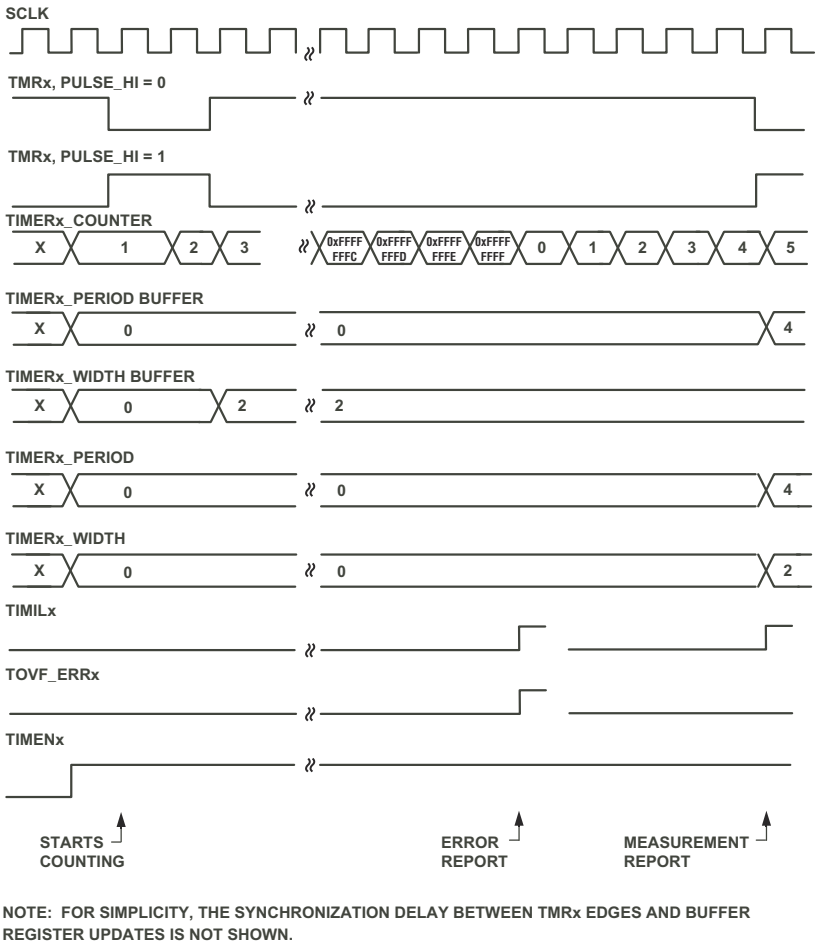
NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 16-18. Example of Width Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 0)

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMERx_PERIOD value in this case returns 0, as shown in Figure 16-18. To measure the pulse width of a waveform that has only one leading edge and one trailing edge, set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer register. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both TIMERx_WIDTH and TIMERx_PERIOD read 0 (because no measurement report occurred to copy the value captured in the width buffer register to TIMERx_WIDTH). See the first interrupt in Figure 16-19.

When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement and logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the Timer Counter register wraps around from 0xFFFF FFFF to 0 in the absence of a leading edge. At that point, the TOVF_ERRx bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMERx_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMERx_PERIOD and TIMERx_WIDTH registers are never updated at the time an error is signaled. Refer to Figure 16-19 and Figure 16-20 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 16-19. Example Timing for Period Overflow Followed by Period Capture (WDTH_CAP mode, PERIOD_CNT = 1)

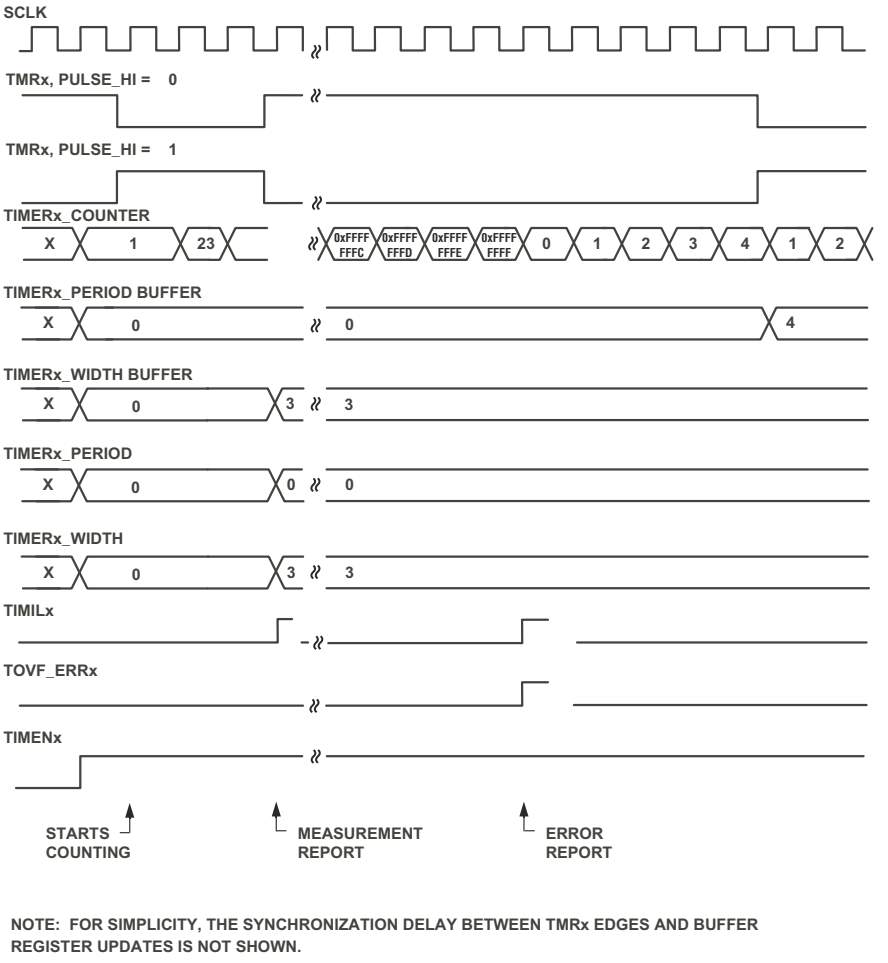


Figure 16-20. Example Timing for Width Capture Followed by Period Overflow (WDTH_CAP mode, PERIOD_CNT = 0)

Both TIMILx and TOVF_ERRx are sticky bits, and software has to explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMERx_PERIOD nor the TIMERx_WIDTH register were updated. If the timer overflowed and PERIOD_CNT = 0, the TIMERx_PERIOD and TIMERx_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 16-19 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure Period = 2 and Pulse Width = 1.

Autobaud Mode

Any one of the three timers may provide autobaud detection for the Universal Asynchronous Receiver/Transmitter (UART0). The Timer Input Select (TIN_SEL) bit in the TIMERx_CONFIG register causes the timer to sample the UART0 port receive data (RX0) pin instead of the TMRx pin when enabled for WDTH_CAP mode.

Do not enable UART0 until after autobaud detection is complete.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with UART0 operation—all derived from the Phase Locked Loop (PLL) clock—the pulse widths can be used to calculate the baud rate divider for UART0.

```
DIVISOR = ((TIMERx_WIDTH) / (16 x Number of captured UART0 bits))
```

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Typically a NULL character (ASCII 0x00) is used in autobaud detection, as shown in Figure 16-21.

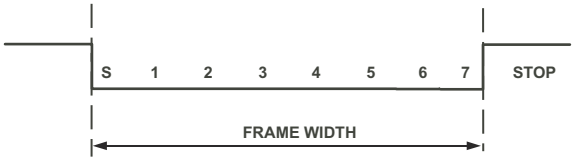


Figure 16-21. Autobaud Detection Character 0x00

Because the example frame in Figure 16-21 encloses 8 data bits and 1 start bit, apply the formula:

```
DIVISOR = TIMERx_WIDTH/(16 x 9)
```

Real UART0 RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character “@” (40h) as an autobaud detection byte and measure the period between two subsequent falling edges. As shown in Figure 16-22, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses 8 bits, apply the formula:

```
DIVISOR = TIMERx_PERIOD/(16 x 8)
```

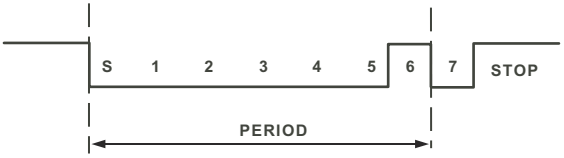


Figure 16-22. Autobaud Detection Character 0x40

External Event (EXT_CLK) Mode

In EXT_CLK mode, the TMRx pin is an input. The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMERx_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMERx_CONFIG register enables this mode. The TIMERx_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period, and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

After the timer has been enabled, it resets the Timer Counter register to 0x0 and then waits for the first leading edge on the TMRx pin. This edge causes the Timer Counter register to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMILx bit is set, and an interrupt is generated. The next leading edge reloads the Timer Counter register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

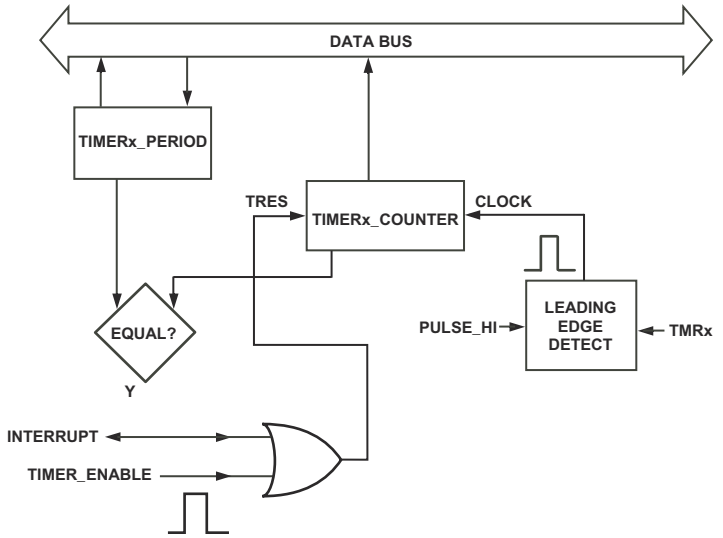


Figure 16-23. Timer Flow Diagram, EXT_CLK Mode

The configuration bits, `TIN_SEL` and `PERIOD_CNT`, have no effect in this mode. The `TOVF_ERRx` and `ERR_TYP` bits are set if the Timer Counter register wraps around from 0xFFFF FFFF to 0 or if `Period = 0` at startup or when the Timer Counter register rolls over (from `Count = Period` to `Count = 0x1`). The Timer Pulse Width register is unused.

Using the Timers With the PPI

Up to two timers are used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to "Frame Synchronization in GP Modes" on page 11-28 of the PPI chapter.

Interrupts

Each of the three timers can generate a single interrupt. The three resulting interrupt signals are routed to the system interrupt controllers block for prioritization and masking. The Timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source. These bits are W1C and must be cleared prior to a RTI to assure that the interrupt is not reissued.

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the System interrupt Mask register (SIC_IMASKx). To poll the TIMILx bit without interrupt generation, set IRQ_ENA but leave the interrupt masked. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions.

The system interrupt controllers enable flexible interrupt handling. All timers may or may not share the same interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, more timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMILx latch bits at once by writing 0x07 to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMILx bit in the TIMERx_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMILx clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMILx bit in the TIMERx_STATUS register at the very beginning of the interrupt service routine (ISR) to avoid missing any timer events.

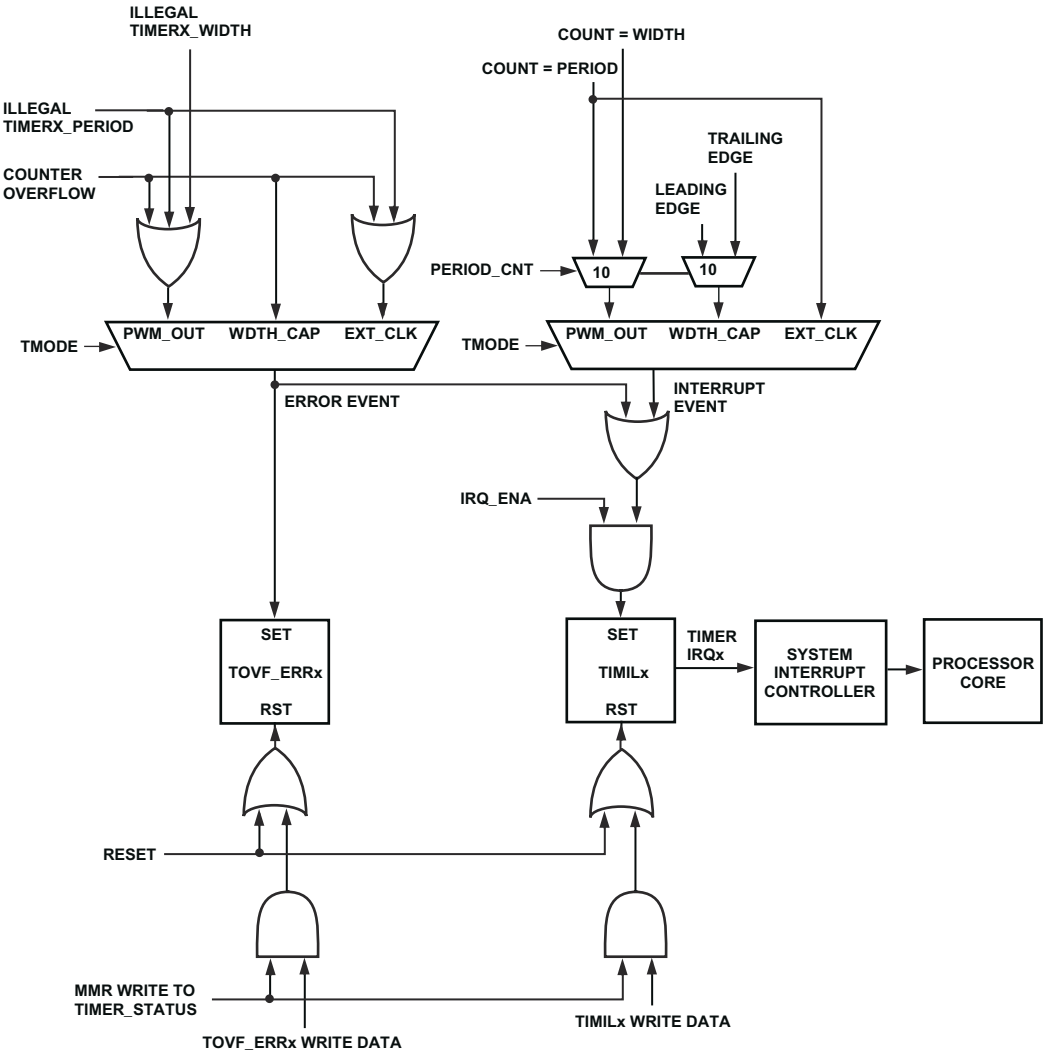


Figure 16-24. Timers Interrupt Structure

Illegal States

For Table 16-1, these definitions are used:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMERx_PERIOD and the counter is reloaded with the value 1.
- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or b#00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads 0 if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads 1.

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write 1 to clear TOVF_ERR to acknowledge the error.

Table 16-1 on page 16-43 can be read as: “In mode __ at event __, if TIMERx_PERIOD is __ and TIMERx_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __.”

Table 16-1. Overview of Illegal States

Mode	Event	TIMERx_ PERIOD	TIMERx_ WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_ CNT = 1	Startup (No boundary condition tests performed on TIMERx_ WIDTH)	== 0	Anything	b#10	Set
		== 1	Anything	b#10	Set
		>= 2	Anything	Unchanged	Unchanged
	Rollover	== 0	Anything	b#10	Set
		== 1	Anything	b#11	Set
		>= 2	== 0	b#11	Set
		>= 2	< TIMERx_ PERIOD	Unchanged	Unchanged
		>= 2	>= TIMERx_ PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMERx_ PERIOD == 0.	Anything	Anything	b#01	Set

Table 16-1. Overview of Illegal States (Cont'd)

Mode	Event	TIMERx_PERIOD	TIMERx_WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_CNT = 0	Startup	Anything	== 0	b#01	Set
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		Anything	>= 1	Unchanged	Unchanged
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMERx_WIDTH == 0.	Anything	Anything	b#01	Set
WDTH_CAP	Startup	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Rollover	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Overflow, not possible unless there is also another error, such as TIMERx_PERIOD == 0.	Anything	Anything	b#01	Set

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMRx pin.

Summary

Table 16-2 summarizes control bit and register usage in each timer mode.

Table 16-2. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - interrupt after measuring period 0 - interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count PPI_CLKs 0 - Count PF1 clocks If CLK_SEL = 0, Unused	1 - Select RX input 0 - Select TMRx input	Unused
OUT_DIS	1 - Disable TMRx pin 0 - Enable TMRx pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused

Table 16-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on event

Table 16-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TRUNx	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect
TOVF_ERR	Set at startup or rollover if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or rollover if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Core Timer

The Core timer is a programmable interval timer which can generate periodic interrupts. The Core timer runs at the core clock (CCLK) rate. The timer includes four core memory mapped Registers (MMRs), the Timer control register (TCNTL), the Timer Count register (TCOUNT), the Timer Period register (TPERIOD), and the Timer Scale register (TSCALE).

Figure 16-25 provides a block diagram of the Core timer.

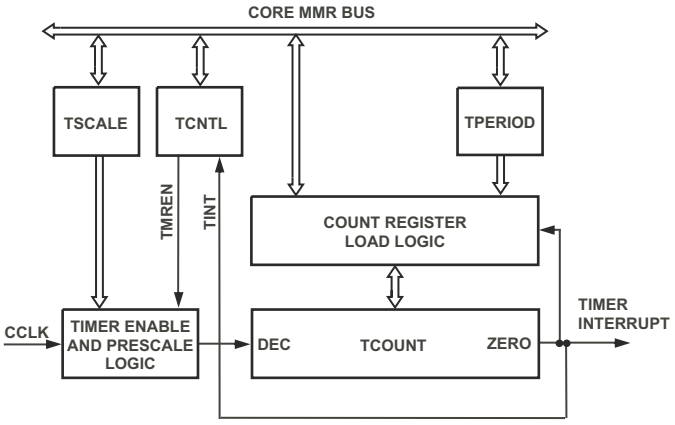


Figure 16-25. Core Timer Block Diagram

TCNTL Register

When the timer is enabled by setting the TMREN bit in the Core Timer control register (TCNTL), the TCOUNT register is decremented once every TSCALE + 1 number of clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register. If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is not a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core timer module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

The Core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR is set, the Core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN is set when TMPWR = 0.

Core Timer Control Register (TCNTL)

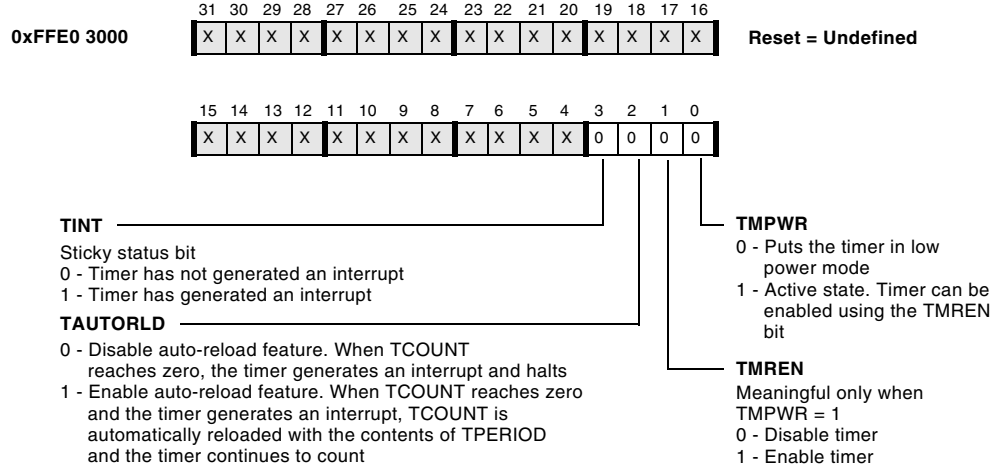


Figure 16-26. Core Timer Control Register

TCOUNT Register

The Core Timer Count register (TCOUNT) decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

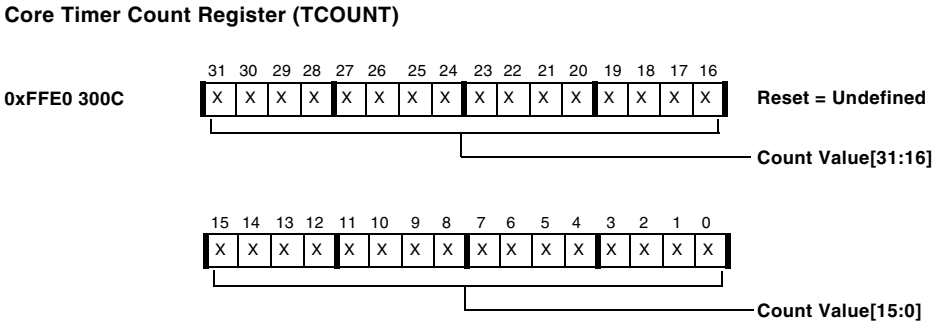


Figure 16-27. Core Timer Count Register

TPERIOD Register

When auto-reload is enabled, the TCOUNT register is reloaded with the value of the Core Timer Period register (TPERIOD) whenever TCOUNT reaches 0.

To ensure that there is valid data in the TPERIOD register, the TPERIOD and TCOUNT registers are initialized simultaneously on the first write to either register. If a different value is desired for the first count period, write the data to TCOUNT after writing to TPERIOD

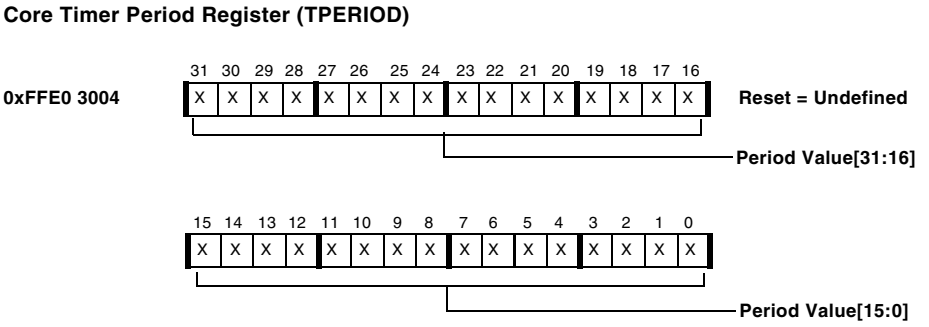


Figure 16-28. Core Timer Period Register

TSCALE Register

The Core Timer Scale register (TSCALE) stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

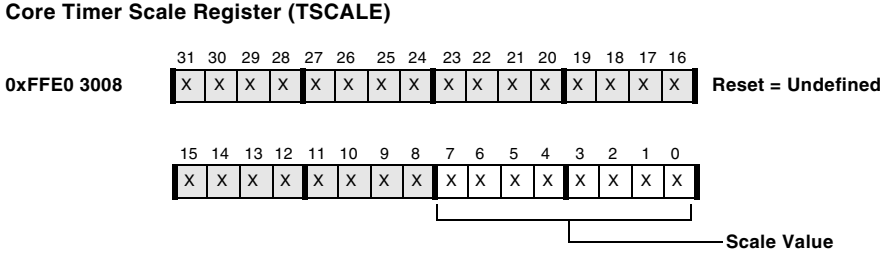


Figure 16-29. Core Timer Scale Register

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the timer expires before being updated by software. Depending on how the watchdog timer is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt. The watchdog timer is clocked by the system clock (SCLK).

Watchdog Timer Operation

To use the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the Watchdog Count register (WDOG_CNT). Note that loading the WDOG_CNT register while the watchdog timer is not enabled will also pre-load the WDOG_STAT register.
2. In the Watchdog control register (WDOG_CTL), select the event to be generated upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register. When the WDOG_STAT reaches 0, the programmed event is generated. To prevent the event from being generated, software must reload the count value from WDOG_CNT to WDOG_STAT by executing a write (of any value) to WDOG_STAT, or must disable the watchdog timer in WDOG_CTL before the watchdog timer expires.

WDOG_CNT Register

The Watchdog Count register (WDOG_CNT) holds the 32-bit unsigned count value. The WDOG_CNT register must be accessed with 32-bit read/writes only.

The Watchdog Count register holds the programmable count value. A valid write to the Watchdog Count register also preloads the Watchdog counter. For added safety, the Watchdog Count register can be updated only when the watchdog timer is disabled. A write to the Watchdog Count register while the timer is enabled does not modify the contents of this register.

Watchdog Count Register (WDOG_CNT)

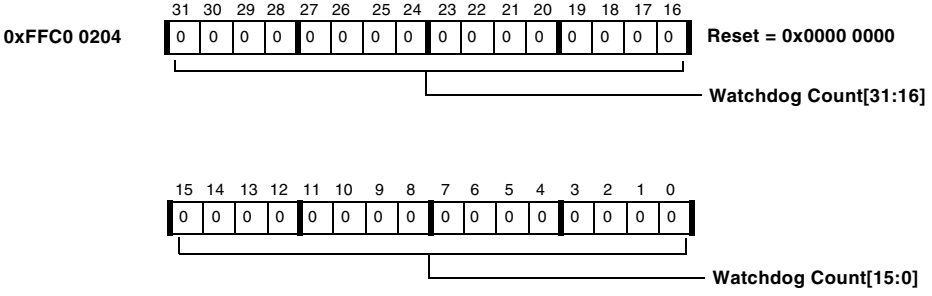


Figure 16-30. Watchdog Count Register

WDOG_STAT Register

The 32-bit Watchdog status register (WDOG_STAT) contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. When the watchdog timer is enabled, WDOG_STAT is decremented by 1 on each SCLK cycle. When WDOG_STAT reaches 0, the watchdog timer stops counting and the event selected in the Watchdog control register (WDOG_CTL) is generated.

The Watchdog Reset event does not reset the Dynamic Power Management Controller (DPMC). A Watchdog Reset that occurs during Sleep Mode will produce undesired results; therefore, the Watchdog should be disabled or reconfigured for a general-purpose interrupt prior to placing the processor into Sleep Mode.

Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, writing the WDOG_STAT register loads it with the value in WDOG_CNT.

When the processor executes a write (of an arbitrary value) to WDOG_STAT, the value in WDOG_CNT is copied into WDOG_STAT. Typically, software sets the value of WDOG_CNT at initialization, then periodically writes to WDOG_STAT before the watchdog timer expires. This reloads the watchdog timer with the value from WDOG_CNT and prevents generation of the selected event.

The WDOG_STAT register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

If the user does not reload the counter before SCLK * Count register cycles, a Watchdog interrupt or reset is generated and the TRO bit in the Watchdog control register is set. When this happens the counter stops decrementing and remains at zero.

If the counter is enabled with a zero loaded to the counter, the TRO bit of the Watchdog control register is set immediately and the counter remains at zero and does not decrement.

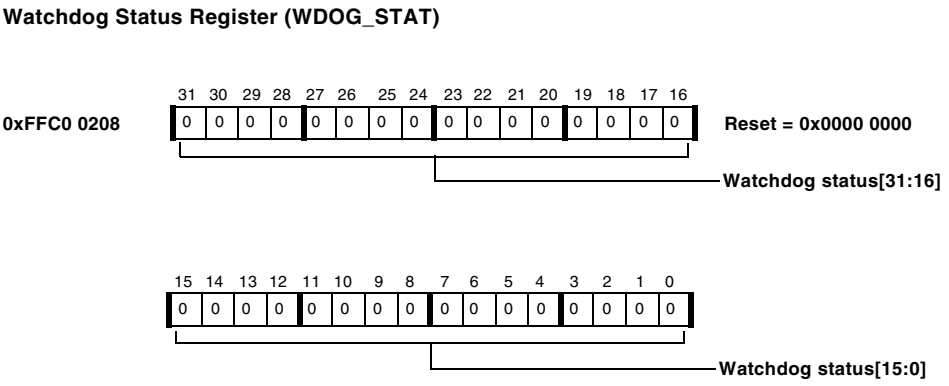


Figure 16-31. Watchdog Status Register

WDOG_CTL Register

The Watchdog control register (WDOG_CTL) is a 16-bit system memory-mapped register used to control the watchdog timer.

The ICTL[1:0] field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the System interrupt Mask register (SIC_IMASK0) should be appropriately configured to unmask that interrupt by setting bit 23. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The TMR_EN[7:0] field is used to enable and disable the watchdog timer. Writing any value other than the disable value into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the timer has rolled over by interrogating the TRO status bit of the Watchdog control register. This is a sticky bit that is set whenever the watchdog timer count reaches 0 and cleared only by disabling the watchdog timer and then writing a 1 to the bit.

Watchdog Control Register (WDOG_CTL)

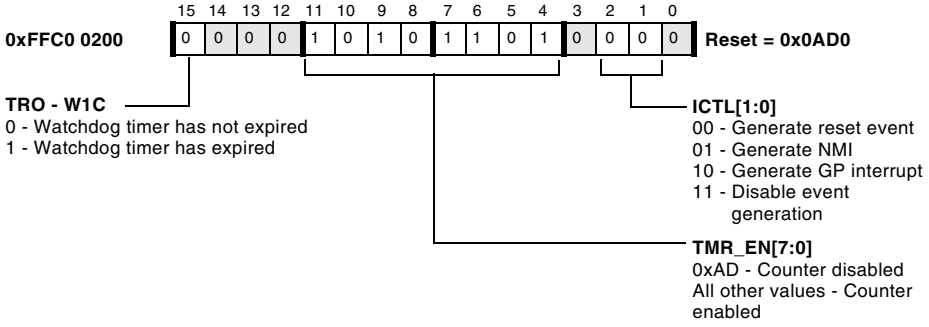


Figure 16-32. Watchdog Control Register

Note that when the processor is in Emulation mode, the watchdog timer counter will not decrement even if it is enabled.

17 REAL-TIME CLOCK

The Real-Time Clock (RTC) provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter.

The RTC watch features are clocked by a 32.768 kHz crystal external to the processor. The RTC uses dedicated power supply pins and is independent of any reset, which enables it to maintain functionality even when the rest of the processor is powered down.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60-second counter
- 60-minute counter
- 24-hour counter
- 32768-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 32768-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.

The RTC provides two alarm features, programmed with the RTC Alarm register (RTC_ALARM). The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC Stopwatch Count register (RTC_SWCNT). When the stopwatch interrupt is enabled and the specified number of seconds have elapsed, the RTC generates an interrupt.

Interfaces

The RTC external interface consists of two clock pins, which together with the external components form the reference clock circuit for the RTC. The RTC interfaces internally to the processor system through the Peripheral Access bus (PAB), and through the interrupt interface to the system interrupt controller (SIC).

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off.

RTC Clock Requirements

The RTC timer is clocked by a 32.768 kHz crystal external to the processor. The RTC system memory-mapped registers (MMRs) are clocked by this crystal. When the prescaler is disabled, the RTC MMRs are clocked at the 32.768 kHz crystal frequency. When the prescaler is enabled, the RTC MMRs are clocked at the 1 Hz rate.

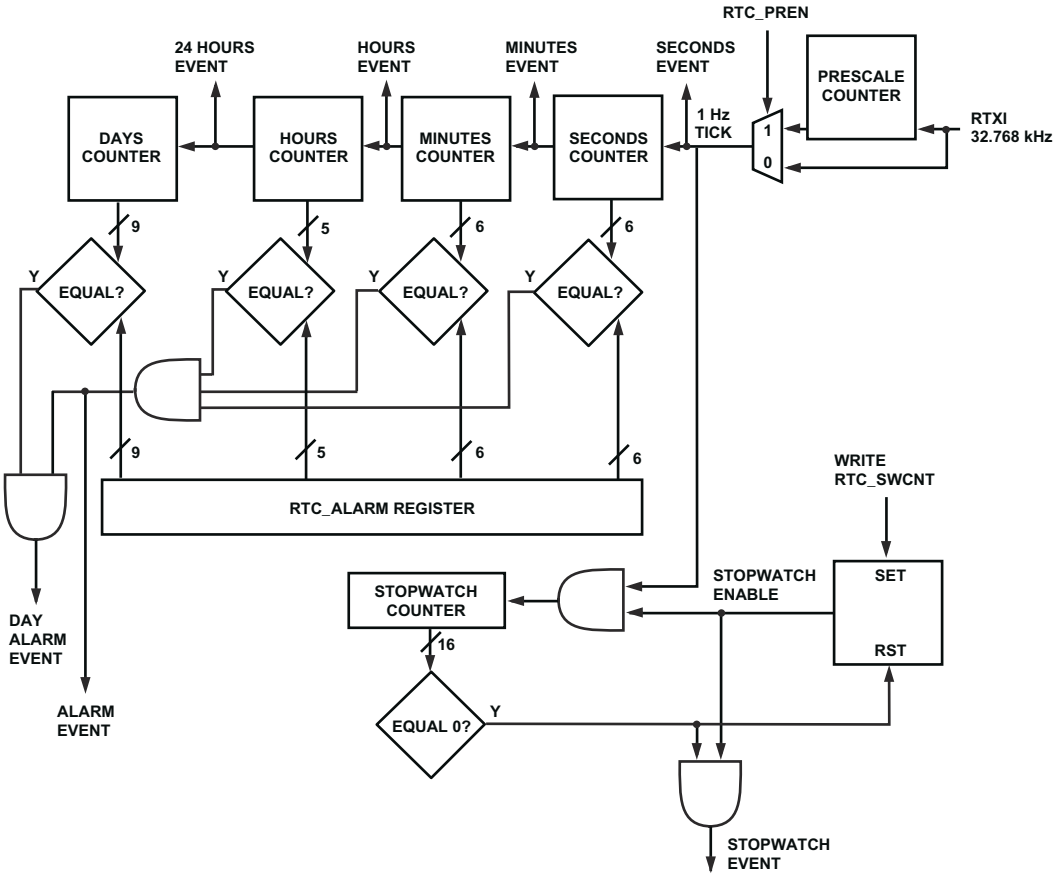


Figure 17-1. RTC Block Diagram

There is no way to disable the RTC counters from software. If a given system does not require the RTC functionality, then it may be disabled with hardware tie-offs. Tie the RTXI pin to EGND, tie the RTCVDD pin to EVDD, and leave the RTXO pin unconnected.

RTC Programming Model

The RTC programming model consists of a set of system MMRs. Software can configure the RTC and can determine the status of the RTC through reads and writes to these registers. The RTC interrupt control register (RTC_ICTL) and the RTC interrupt status register (RTC_ISTAT) provide RTC interrupt management capability.

Note that software cannot disable the RTC counting function. However, all RTC interrupts can be disabled or masked. At reset, all interrupts are disabled. The RTC state can be read via the system MMR status registers at any time.

The primary Real-Time Clock functionality, shown in Figure 17-1 on page -3, consists of registers and counters that are powered by an independent RTC V_{dd} supply. This logic is never reset; it comes up in an unknown state when RTC V_{dd} is first powered on.

The RTC also contains logic powered by the same internal V_{dd} as the processor core and other peripherals. This logic contains some control functionality, holding registers for PAB write data, and prefetched PAB read data shadow registers for each of the five RTC V_{dd}-powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

Figure 17-2 on page -5 shows the connections between the RTC V_{dd}-powered RTC MMRs and their corresponding internal V_{dd}-powered write holding registers and read shadow registers. In the figure, “REG” means each of the RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN registers. The RTC_ISTAT register connects only to the PAB.

The rising edge of the 1 Hz RTC clock is the “1 Hz tick”. Software can synchronize to the 1 Hz tick by waiting for the Seconds Event flag to set or by waiting for the Seconds interrupt (if enabled).

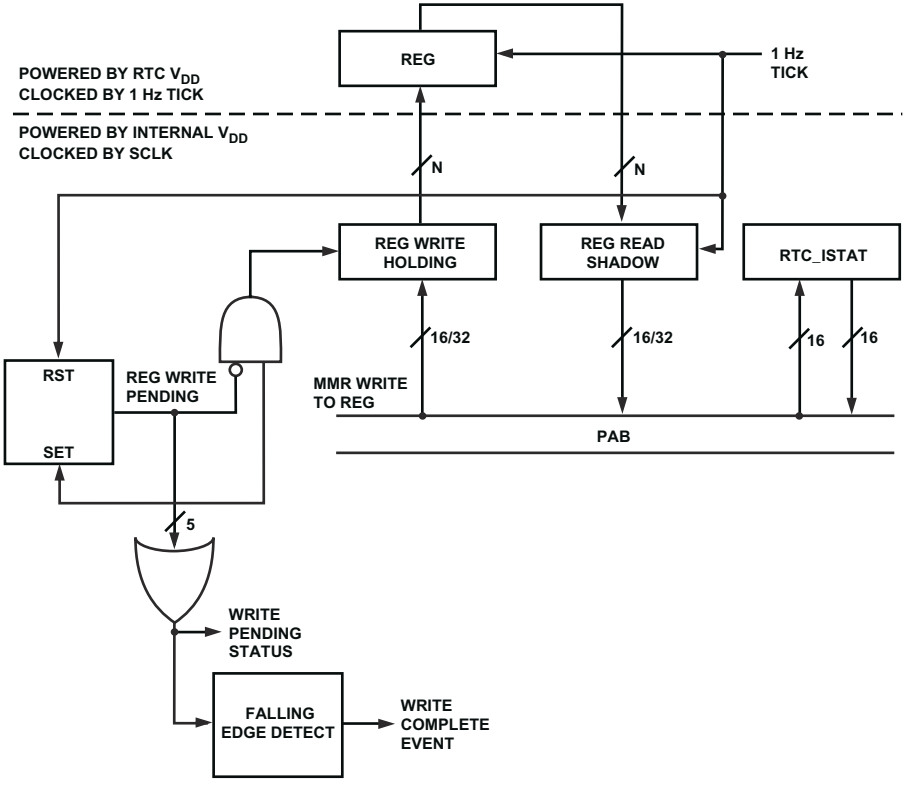


Figure 17-2. RTC Register Architecture

Register Writes

Writes to all RTC MMRs, except the RTC interrupt status register (RTC_ISTAT), are saved in write holding registers and then are synchronized to the RTC 1 Hz clock. The Write Pending status bit in RTC_ISTAT indicates the progress of the write. The Write Pending status bit is set when a write is initiated and is cleared when all writes are complete. The falling edge of the Write Pending status bit causes the Write Complete flag in RTC_ISTAT to be set. This flag can be configured in RTC_ICTL to cause an interrupt. Software does not have to wait for writes to one RTC

MMR to complete before writing to another RTC MMR. The Write Pending status bit is set if any writes are in progress, and the Write Complete flag is set only when all writes are complete.

Any writes in progress when peripherals are reset will be aborted. Do not stop SCLK (enter Deep Sleep mode) or remove Internal V_{dd} power (enter Hibernate state) until all RTC writes have completed.

Do not attempt another write to the same register without waiting for the previous write to complete. Subsequent writes to the same register are ignored if the previous write is not complete.

Reading a register that has been written before the Write Complete flag is set will return the old value. Always check the Write Pending status bit before attempting a read or write.

Write Latency

Writes to the RTC MMRs (except RTC_ISTAT) are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the Real-Time Clock is to monitor the Seconds (1 Hz) Event flag or to program an interrupt for this event and then write the current time to the RTC status register (RTC_STAT) in the interrupt service routine (ISR). The new value is inserted ahead of the incrementer. Hardware adds one second to the written value (with appropriate carries into minutes, hours and days) and loads the incremented value at the next 1 Hz tick, when it represents the then-current time.

Writes posted at any time are properly synchronized to the 1 Hz clock. Writes complete at the rising edge of the 1 Hz clock. A write posted just before the 1 Hz tick may not be completed until the 1 Hz tick one second later. Any write posted in the first 990 ms after a 1 Hz tick will complete on the next 1 Hz tick, but the simplest, most predictable and recommended technique is to only post writes to RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, or RTC_PREN immediately after a Seconds interrupt or Event. All five registers may be written in the same second.

W1C bits in the RTC_ISTAT register take effect immediately.

Register Reads

There is no latency when reading RTC MMRs, as the values come from the read shadow registers. The shadows are updated and ready for reading by the time any RTC interrupts or Event flags for that second are asserted. Once the internal V_{dd} logic completes its initialization sequence after SCLK starts, there is no point in time when it is unsafe to read the RTC MMRs for synchronization reasons. They always return coherent values, although the values may be unknown.

Deep Sleep

When the dynamic power management controller (DPMC) state is deep sleep, all clocks in the system (except RTXI and the RTC 1 Hz tick) are stopped. In this state, the RTC V_{dd} counters continue to increment. The internal V_{dd} shadow registers are not updated and cannot be read.

During deep sleep mode, all bits in RTC_ISTAT are cleared. Events that occur during deep sleep are not recorded in RTC_ISTAT. The internal V_{dd} RTC control logic generates a virtual 1 Hz tick within one RTXI period (30.52 µs) after SCLK restarts. This loads all shadow registers with up-to-date values and sets the seconds event flag. Other Event flags may

also be set. When the system wakes up from deep sleep, whether by an RTC event or a hardware reset, all of the RTC events that occurred during that second (and only that second) are reported in RTC_ISTAT.

When the system wakes up from deep sleep mode, software does not need to W1C the bits in RTC_ISTAT. All “write 1 to clear” bits are already cleared by hardware. The seconds event flag is set when the RTC internal V_{dd} logic has completed its restart sequence. Software should wait until the seconds event flag is set and then may begin reading or writing any RTC register.

Prescaler Enable

The single active bit of the RTC prescaler enable register (RTC_PREN) is written using a synchronization path. Clearing of the bit is synchronized to the 32.768 kHz clock. This faster synchronization allows the module to be put into high-speed mode (bypassing the prescaler) without waiting the full 1 second for the write to complete that would be necessary if the module were already running with the prescaler enabled.

When setting the RTC_PREN bit, the first positive edge of the 1 Hz clock occurs 1 to 2 cycles of the 32.768 kHz clock after the prescaler is enabled. The Write Complete status/interrupt works as usual when enabling or disabling the prescale counter. The new RTC clock rate is in effect before the Write Complete status is set.

Event Flags

The unknown values in the registers at powerup can cause Event flags to set before the correct value is written into each of the registers. By catching the 1 Hz clock edge, the write to RTC_STAT can occur a full second before the write to RTC_ALARM. This would cause an extra second of delay between the validity of RTC_STAT and RTC_ALARM, if the value of the RTC_ALARM out of reset is the same as the value written to RTC_STAT. Wait for the writes to complete on these registers before using the flags and interrupts associated with their values.

The following is a list of flags along with the conditions under which they are valid:

- Seconds (1 Hz) Event flag

  Always set on the positive edge of the 1 Hz clock and after shadow registers have updated after waking from Deep Sleep. This is valid as long as the RTC 1 Hz clock is running. Use this flag or interrupt to validate the other flags.

- Write Complete

  Always valid.

- Write Pending status

  Always valid.

- Minutes Event flag

  Valid only after the seconds field in RTC_STAT is valid. Use the Write Complete and Write Pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- Hours Event flag

  Valid only after the minutes field in RTC_STAT is valid. Use the Write Complete and Write Pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- 24 Hours Event flag

  Valid only after the hours field in RTC_STAT is valid. Use the Write Complete and Write Pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- Stopwatch Event flag

  Valid only after the RTC_SWCNT register is valid. Use the Write Complete and Write Pending status flags or interrupts to validate the RTC_SWCNT value before using this flag value or enabling the interrupt.

- Alarm Event flag

  Valid only after the RTC_STAT and RTC_ALARM registers are valid. Use the Write Complete and Write Pending status flags or interrupts to validate the RTC_STAT and RTC_ALARM values before using this flag value or enabling its interrupt.

- Day Alarm Event flag

  Same as Alarm.

Writes posted together at the beginning of the same second take effect together at the next 1 Hz tick. The following sequence is safe and does not result in any spurious interrupts from a previous state.

1. Wait for 1 Hz tick.
2. Write 1s to clear the RTC_ISTAT flags for Alarm, Day Alarm, Stopwatch, and/or per-interval.
3. Write new values for RTC_STAT, RTC_ALARM, and/or RTC_SWCNT.
4. Write new value for RTC_ICTL with Alarm, Day Alarm, Stopwatch, and/or per-interval interrupts enabled.
5. Wait for 1 Hz tick.
6. New values have now taken effect simultaneously.

Interrupts

The RTC can provide interrupts at several programmable intervals, including:

- Per second
- Per minute
- Per hour
- Per day
- On countdown from a programmable value
- Daily at a specific time
- On a specific day and time

The RTC can be programmed to provide an interrupt at the completion of all pending writes to any of the 1 Hz registers (RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN). Interrupts can be individually enabled or disabled using the RTC interrupt control register (RTC_ICTL). Interrupt status can be determined by reading the RTC interrupt status register (RTC_ISTAT).

The RTC interrupt is set whenever an event latched into the RTC_ISTAT register is enabled in the RTC_ICTL register. The pending RTC interrupt is cleared whenever all enabled and set bits in RTC_ISTAT are cleared, or when all bits in RTC_ICTL corresponding to pending events are cleared.

As shown in Figure 17-3 on page -13, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wakeup from a Sleep state. The RTC generates a separate signal for wakeup from Deep Sleep or from an internal V_{dd} power-off state. The Deep Sleep wakeup signal is asserted at the 1 Hz tick when any RTC interval event enabled in RTC_ICTL occurs. The assertion of the Deep Sleep wakeup signal causes the processor core clock (CCLK) and the system clock (SCLK) to restart. Any enabled event that asserts the RTC Deep Sleep wakeup signal also causes the RTC IRQ to assert once SCLK restarts.

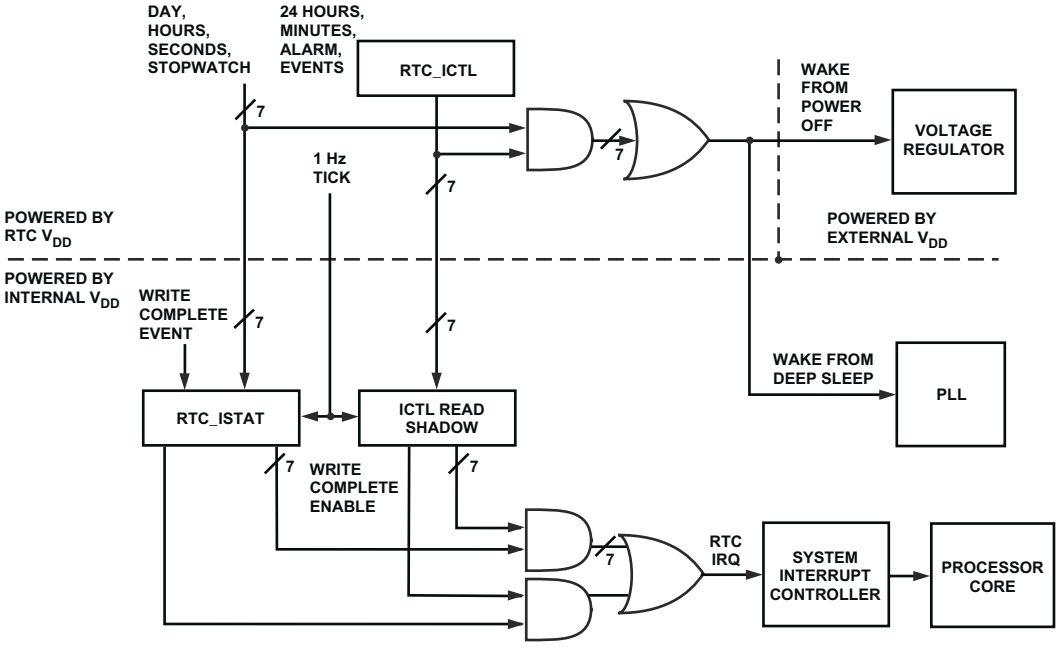


Figure 17-3. RTC Interrupt Structure

RTC_STAT Register

The RTC status register (RTC_STAT) is used to read or write the current time. Reads return a 32-bit value that always reflects the current state of the days, hours, minutes, and seconds counters. Reads and writes must be 32-bit transactions; attempted 16-bit transactions result in an MMR error. Reads always return a coherent 32-bit value. The hours, minutes, and seconds fields are usually set to match the real time of day. The day counter value is incremented every day at midnight to record how many days have elapsed since it was last modified. Its value does not correspond to a particular calendar day. The 15-bit day counter provides a range of 89 years, 260 or 261 days (depending on leap years) before it overflows.

After the 1 Hz tick, program RTC_STAT with the current time. At the next 1 Hz tick, RTC_STAT takes on the new, incremented value. For example:

1. Wait for 1 Hz tick.
2. Read RTC_STAT, get 10:45:30.
3. Write RTC_STAT to current time, 13:10:59.
4. Read RTC_STAT, still get old time 10:45:30.
5. Wait for 1 Hz tick.
6. Read RTC_STAT, get new current time, 13:11:00.

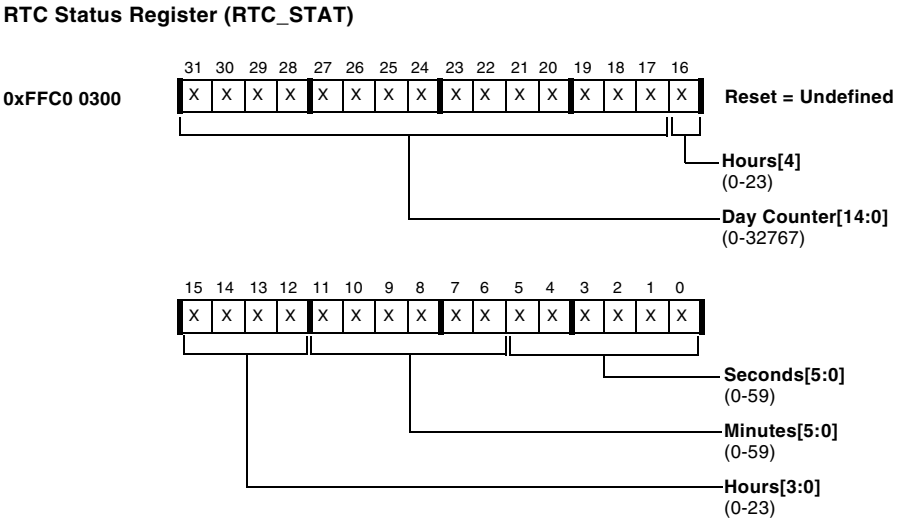


Figure 17-4. RTC Status Register

RTC_ICTL Register

The eight RTC interrupt events can be individually masked or enabled by the RTC interrupt control register (RTC_ICTL). The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24-hour interrupt occurs once per 24-hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wakeup request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC interrupt and the RTC Wakeup to the PLL are handled specially and are masked (forced low) until after the first write to the RTC_ICTL register is complete. Therefore, all interrupts act as if they were disabled at system reset (as if all bits of RTC_ICTL were zero), even thought some bits of RTC_ICTL may read as nonzero. If no RTC interrupts are needed immediately after reset, it is recommended to write RTC_ICTL to 0x0000 so that later read-modify-write accesses will function as intended.

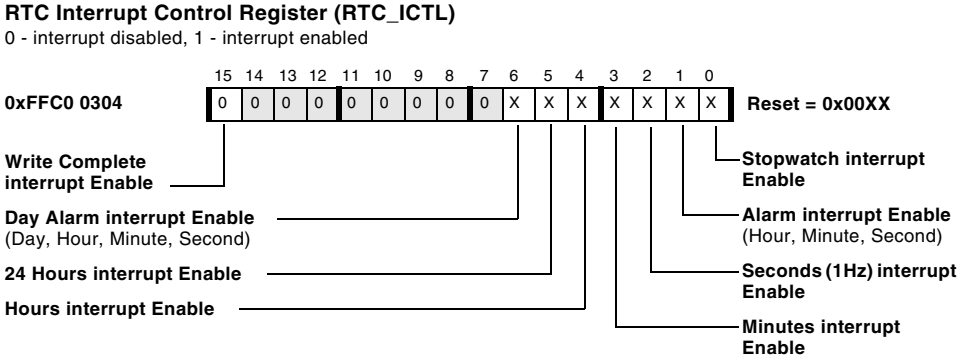


Figure 17-5. RTC Interrupt Control Register

RTC_ISTAT Register

The RTC interrupt status register (RTC_ISTAT) provides the status of all RTC interrupts. These bits are sticky. Once set by the corresponding event, each bit remains set until cleared by a software write to this register. Event flags are always set; they are not masked by the interrupt enable bits in RTC_ICTL. Values are cleared by writing a 1 to the respective bit location, except for the Write Pending status bit, which is read-only. Writes of 0 to any bit of the register have no effect. This register is cleared at reset and during Deep Sleep.

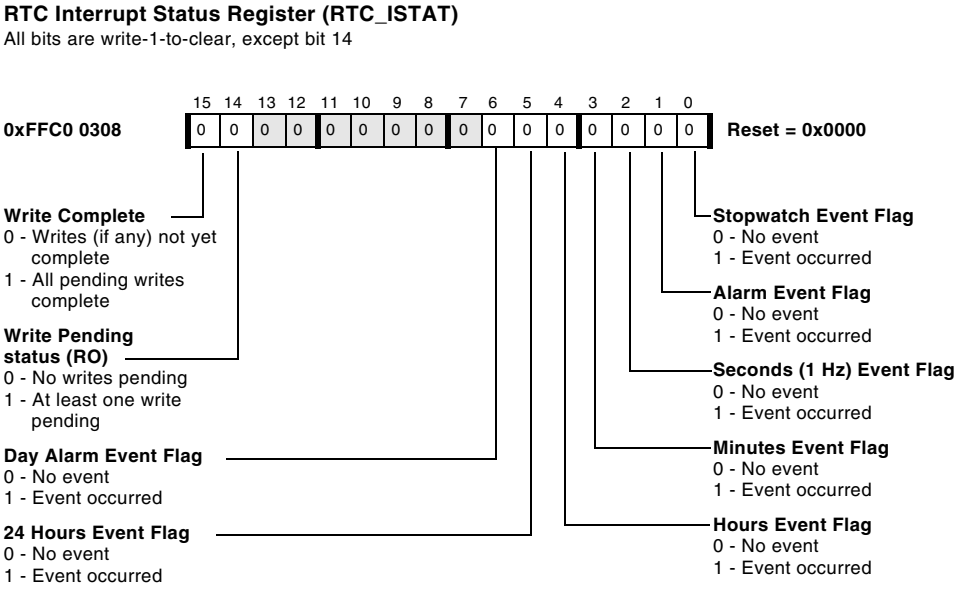


Figure 17-6. RTC Interrupt Status Register

RTC_SWCNT Register

The RTC Stopwatch Count register (RTC_SWCNT) contains the countdown value for the stopwatch. The stopwatch counts down seconds from the programmed value and generates an interrupt (if enabled) when the count reaches 0. The counter stops counting at this point and does not resume counting until a new value is written to RTC_SWCNT. Once running, the counter may be overwritten with a new value. This allows the stopwatch to be used as a watchdog timer with a precision of one second. Writing the running stopwatch to 0 forces it to stop and interrupt early. The Stopwatch Event flag is set at the 1 Hz tick at which any of these occur:

- The stopwatch counter decrements to 0x0000
- A write of 0x0000 to RTC_SWCNT completes and the stopwatch was running (current stopwatch count was greater than 0)
- A write of 0x0000 to RTC_SWCNT completes and the stopwatch was stopped (current stopwatch count was equal to 0)

The register can be programmed to any value between 0 and (2^{16} – 1) seconds. This is a range of 18 hours, 12 minutes, and 15 seconds.

Typically, software should wait for a 1 Hz tick, then write RTC_SWCNT. One second later, RTC_SWCNT changes to the new value and begins decrementing. Because the register write occupies nearly one second, the time from writing a value of N until the stopwatch interrupt is nearly N + 1 seconds. To produce an exact delay, software can compensate by writing N - 1 to get a delay of nearly N seconds. This implies that you cannot achieve a delay of 1 second with the stopwatch. Writing a value of 1 immediately after a 1 Hz tick results in a stopwatch interrupt nearly two seconds later. To wait one second, software should just wait for the next 1 Hz tick.

The RTC Stopwatch Count register is not reset. After initial powerup, it may be running. When the stopwatch is not used, writing it to 0 to force it to stop saves a small amount of power.

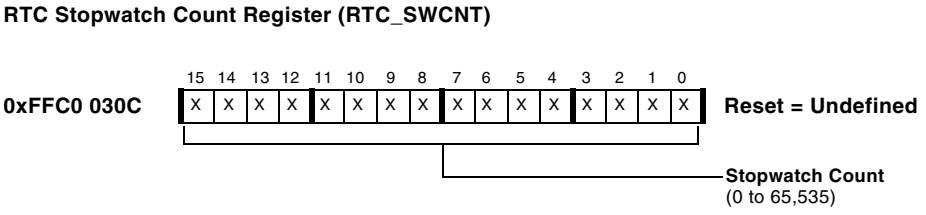


Figure 17-7. RTC Stopwatch Count Register

RTC_ALARM Register

The RTC Alarm register (RTC_ALARM) is programmed by software for the time (in hours, minutes, and seconds) the alarm interrupt occurs. Reads and writes can occur at any time. The alarm interrupt occurs whenever the hour, minute, and second fields first match those of the RTC status register. The day interrupt occurs whenever the day, hour, minute, and second fields first match those of the RTC status register.

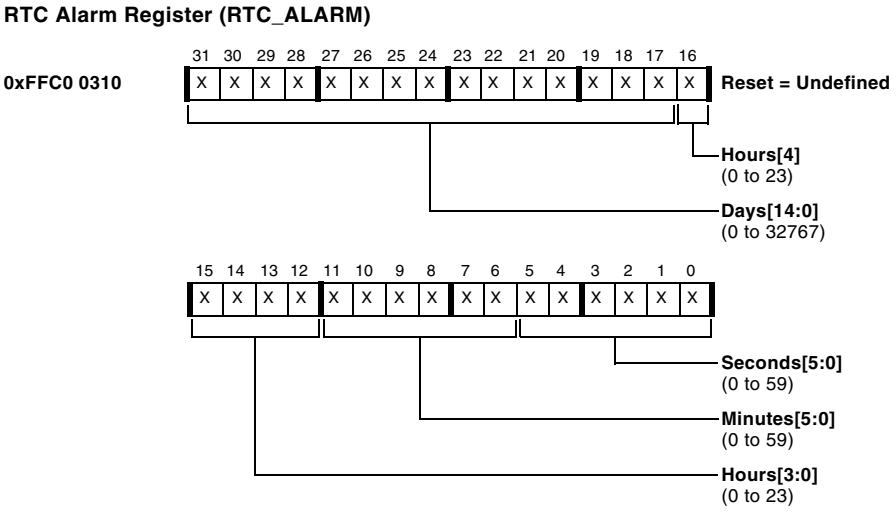


Figure 17-8. RTC Alarm Register

RTC_PREN Register

The RTC Prescaler Enable register (RTC_PREN) has one active bit. When this bit is set, the prescaler is enabled, and the RTC runs at a frequency of 1 Hz. When this bit is cleared, the prescaler is disabled, and the RTC runs at the 32.768 kHz crystal frequency.

In order for the RTC to operate at the proper rate, software must set the Prescaler Enable bit after initial powerup. Write RTC_PREN and then wait for the Write Complete event before programming the other registers. It is safe to write RTC_PREN to 1 every time the processor boots. The first time sets the bit, and subsequent writes will have no effect, as no state is changed.

Do not disable the prescaler by clearing the bit in RTC_PREN without making sure that there are no writes to RTC MMRs in progress. Do not switch between fast and slow mode during normal operation by setting and clearing this bit, as this disrupts the accurate tracking of real time by the counters. To avoid these potential errors, initialize the RTC during startup via RTC_PREN and do not dynamically alter the state of the prescaler during normal operation.

Running without the prescaler enabled is provided primarily as a test mode. All functionality works, just 32,768 times as fast. Typical software should never program RTC_PREN to 0. The only reason to do so is to synchronize the 1 Hz tick to a more precise external event, as the 1 Hz tick predictably occurs a few RTXI cycles after a 0 → 1 transition of RTC_PREN. Use the following sequence to achieve synchronization to within 100 µs.

1. Write RTC_PREN to 0.
2. Wait for the write to complete.
3. Wait for the external event.
4. Write RTC_PREN to 1.
5. Wait for the write to complete.
6. Reprogram the time into RTC_STAT.

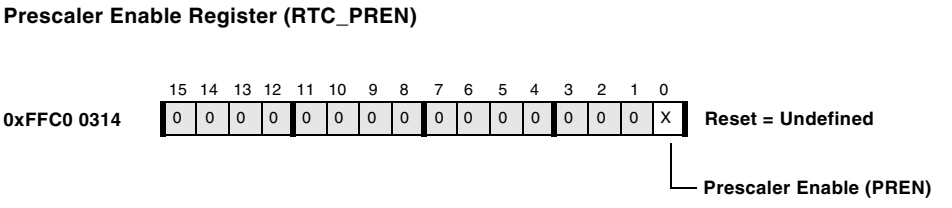


Figure 17-9. Prescaler Enable Register

State Transitions Summary

Table 17-1 shows how each RTC MMR is affected by the system states. The phase locked loop (PLL) states (Reset, Full On, Active, Sleep, and Deep Sleep) are defined in Chapter 8, "Dynamic Power Management" "No Power" means none of the processor power supply pins are connected to a source of energy. "Off" means the processor core, peripherals, and memory are not powered (Internal V_{dd} is off), while the RTC is still powered and running. External V_{dd} may still be powered. Registers described as "As written" are holding the last value software wrote to the register. If the register has not been written since RTC V_{dd} power was applied, then the state is unknown (for all bits of `RTC_STAT`, `RTC_ALARM`, and `RTC_SWCNT`, and for some bits of `RTC_ISTAT`, `RTC_PREN`, and `RTC_ICTL`).

Table 17-1. Effect of States on RTC MMRs

RTC V_{dd}	IV_{dd}	System State	RTC_ICTL	RTC_ISTAT	RTC_STAT RTC_SWCNT	RTC_ALARM RTC_PREN
Off	Off	No Power	X	X	X	X
On	On	Reset	As written	0	Counting	As written
On	On	Full On	As written	Events	Counting	As written
On	On	Sleep	As written	Events	Counting	As written
On	On	Active	As written	Events	Counting	As written
On	On	Deep Sleep	As written	0	Counting	As written
On	Off	Off	As written	X	Counting	As written

Table summarizes software's responsibilities with respect to the RTC at various system state transition events.

Table 17-2. RTC System State Transition Events

At This Event:	Execute This Sequence:
Power On from No Power	Write RTC_PREN = 1. Wait for Write Complete. Write RTC_STAT to current time. Write RTC_ALARM, if needed. Write RTC_SWCNT. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.
Full On after Reset or Full On after Power On from Off	Wait for Seconds Event, or write RTC_PREN = 1 and wait for Write Complete. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts. Read RTC MMRs as required.
Wake from Deep Sleep	Wait for Seconds Event flag to set. Write RTC_ISTAT to acknowledge RTC Deep Sleep wakeup. Read RTC MMRs as required. The PLL state is now Active. Transition to Full On as needed.
Wake from Sleep	If wakeup came from RTC, Seconds Event flag will be set. In this case, write RTC_ISTAT to acknowledge RTC wakeup IRQ. Always, read RTC MMRs as required.
Before Going to Sleep	If wakeup by RTC is desired: Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC interrupt sources for wakeup. Wait for Write Complete. Enable RTC for wakeup in the System interrupt Wakeup-Enable register (SIC_IWR0).

Table 17-2. RTC System State Transition Events (Cont'd)

At This Event:	Execute This Sequence:
Before Going to Deep Sleep	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC event sources for Deep Sleep wakeup. Wait for Write Complete.
Before Going to Hibernate	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable any desired RTC event sources for powerup wakeup. Wait for Write Complete. Set the WAKE bit in the Voltage Regulator control register (VR_CTL).

18 EXTERNAL BUS INTERFACE UNIT

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

Overview

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

The EBIU is clocked by the system clock (`SCLK`). All synchronous memories interfaced to the processor operate at the `SCLK` frequency. The ratio between core frequency and `SCLK` frequency is programmable using a phase-locked loop (PLL) system memory-mapped register (MMR). For more information, see "Core Clock/System Clock Ratio Control" on page 8-4.

The external memory space is shown in Figure 18-1 on page 18-3. One memory region is dedicated to SDRAM support. SDRAM interface timing and the size of the SDRAM region are programmable. The SDRAM memory space can range in size from 16 to 128M byte.

For information on how to connect to SDRAMs smaller than 16M byte, see "Using SDRAMs Smaller than 16M byte" on page 22-7.

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

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

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

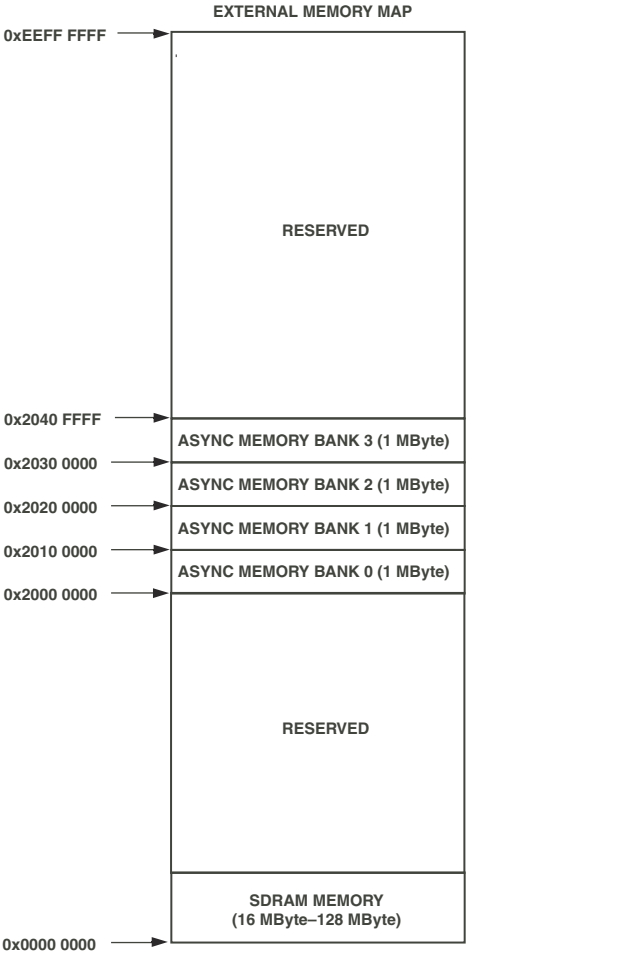


Figure 18-1. External Memory Map

Block Diagram

Figure 18-2 is a conceptual block diagram of the EBIU and its interfaces. Signal names shown with an overbar are active low signals.

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

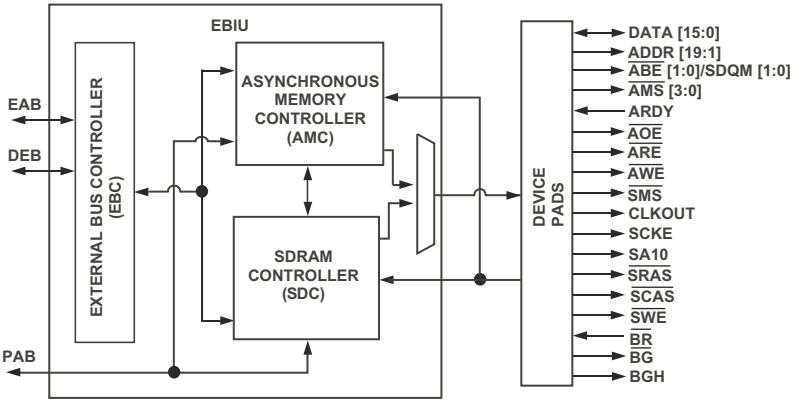


Figure 18-2. External Bus Interface Unit (EBIU)

Internal Memory Interfaces

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

- External Access Bus (EAB), mastered by the core memory management unit on behalf of external bus requests from the core

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

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

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

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

External Memory Interfaces

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

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

No other signals are multiplexed between the two controllers.The following tables describe the signals associated with each interface.

Table 18-1. Asynchronous Memory Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External Data Bus
ADDR[19:1]	O	External Address Bus
$\overline{\text{AMS[3:0]}}$	O	Asynchronous memory Selects
$\overline{\text{AWE}}$	O	Asynchronous memory Write Enable
$\overline{\text{ARE}}$	O	Asynchronous memory Read Enable
$\overline{\text{AOE}}$	O	Asynchronous memory Output Enable In most cases, the $\overline{\text{AOE}}$ pin should be connected to the $\overline{\text{OE}}$ pin of an external memory-mapped asynchronous device. Refer to *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for specific timing information between the $\overline{\text{AOE}}$ and $\overline{\text{ARE}}$ signals to determine which interface signal should be used in your system.
ARDY	I	Asynchronous memory Ready Response Note this is a synchronous input
$\overline{\text{ABE[1:0]}}$/SDQM[1:0]	O	Byte Enables

1 Pin Types: I = Input, O = Output

Table 18-2. SDRAM Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External Data Bus
ADDR[19:18], ADDR[16:1]	O	External Address Bus Connect to SDRAM Address pins. bank address is output on ADDR[19:18] and should be connected to SDRAM BA[1:0] pins.
$\overline{\text{SRAS}}$	O	SDRAM Row Address Strobe pin Connect to the SDRAM $\overline{\text{RAS}}$ pin.
$\overline{\text{SCAS}}$	O	SDRAM Column Address Strobe pin Connect to the SDRAM $\overline{\text{CAS}}$ pin.
$\overline{\text{SWE}}$	O	SDRAM Write Enable pin Connect to the SDRAM $\overline{\text{WE}}$ pin.
$\overline{\text{ABE[1:0]}}$/ SDQM[1:0]	O	SDRAM Data Mask pins Connect to the SDRAM DQM pins.
$\overline{\text{SMS}}$	O	memory Select pin of external memory bank configured for SDRAM Connect to the SDRAM $\overline{\text{CS}}$ (Chip Select) pin. Active Low.
SA10	O	SDRAM A10 pin SDRAM interface uses this pin to be able to do refreshes while the AMC is using the bus. Connect to the SDRAM A[10] pin.
SCKE	O	SDRAM Clock Enable pin Connect to the SDRAM CKE pin.
CLKOUT	O	SDRAM Clock Output pin Switches at system clock frequency. Connect to the SDRAM CLK pin.

1 Pin Types: I = Input, O = Output

EBIU Programming Model

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers used to program the EBIU.

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

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

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

Error Detection

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

- Any access to reserved off-chip memory space
- Any access to a disabled external memory bank
- Any access to an unpopulated area of an SDRAM memory bank

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

Asynchronous Memory Interface

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

Table 18-3. Asynchronous Memory Bank Address Range

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

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 1M byte; however, not all of an enabled memory bank need be populated. Unlike the SDRAM memory, which may need to support very large memory structures spanning multiple memory banks, it should be relatively easy to constrain code and data structures to fit within one of the supported asynchronous memory banks, because of the nature of the types of code or data that is stored here.

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

The asynchronous memory signals are defined in Table 18-1 on page 18-6. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see Chapter 22, “System Design”

EBIU_AMGCTL Register

The asynchronous memory global control register (EBIU_AMGCTL) configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be programmed while the AMC is in use. The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

For external devices that need a clock, CLKOUT can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

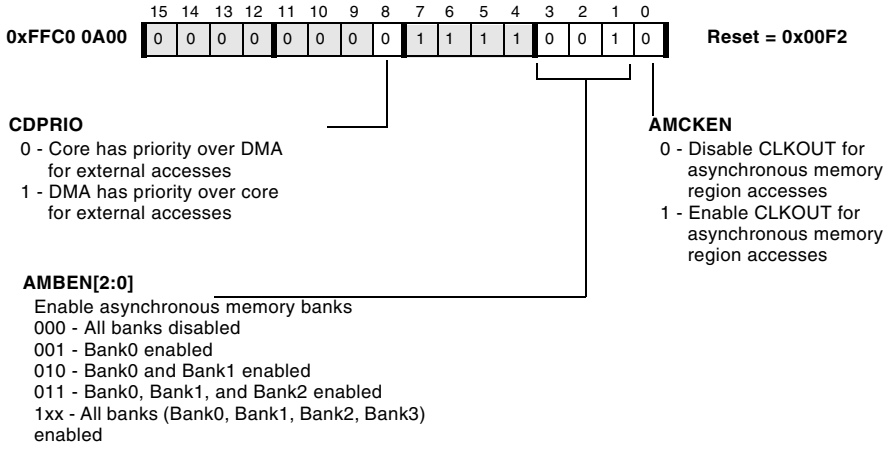


Figure 18-3. Asynchronous Memory Global Control Register

EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers

The EBIU asynchronous memory controller has two asynchronous memory bank control registers (EBIU_AMBCTL0 and EBIU_AMBCTL1). They contain bits for counters for setup, strobe, and hold time; bits to determine memory type and size; and bits to configure use of ARDY. These registers should not be programmed while the AMC is in use.

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

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

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

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

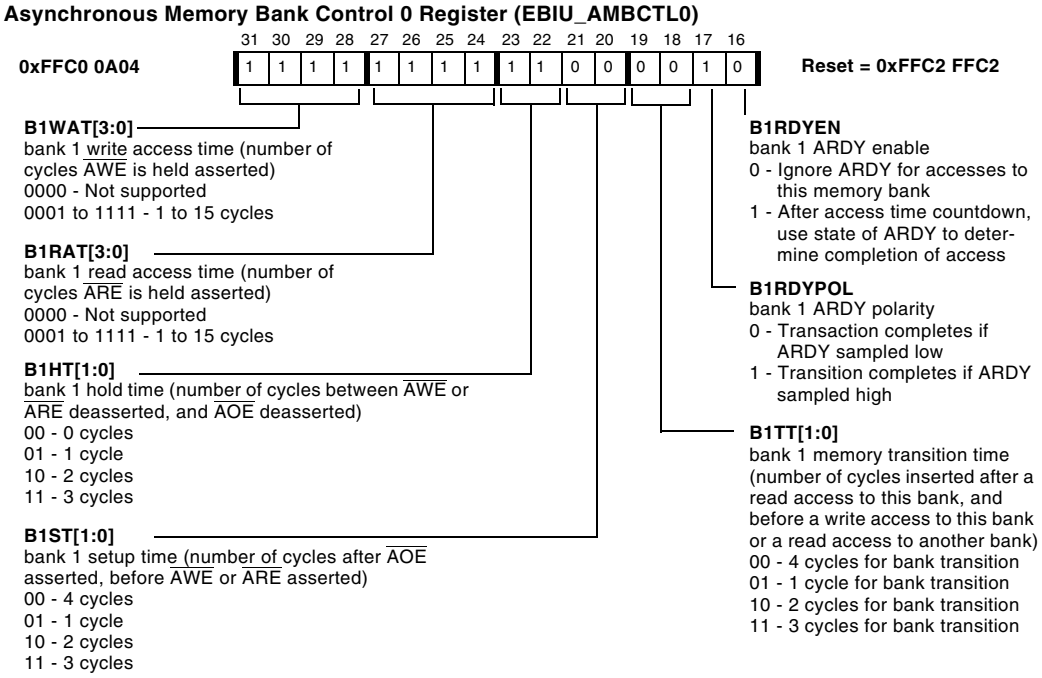


Figure 18-4. Asynchronous Memory Bank Control 0 Register (Bits 31–16)

Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)

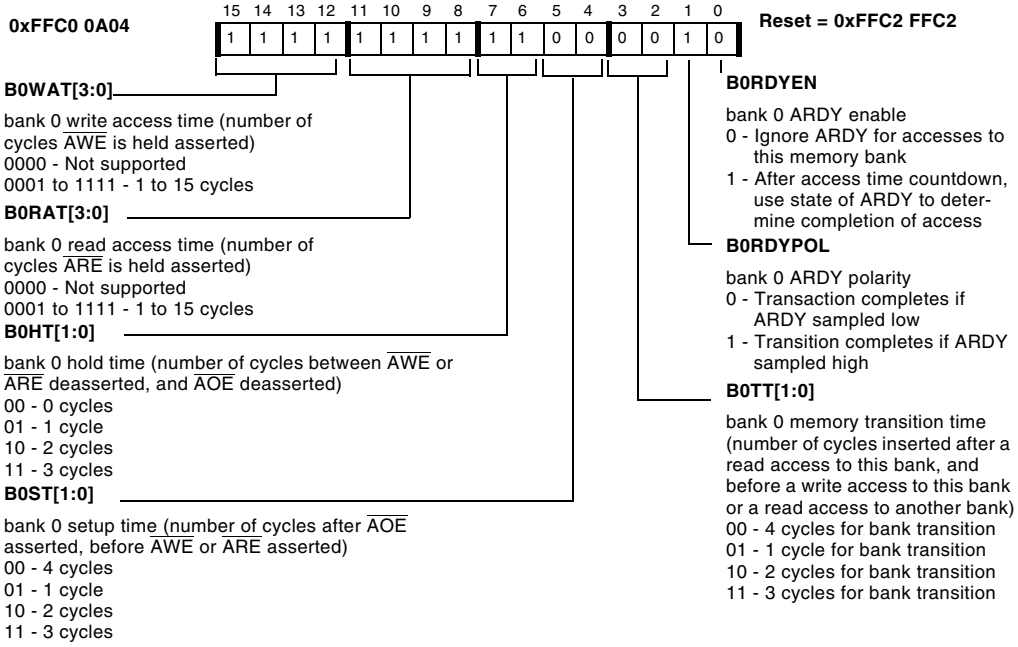


Figure 18-5. Asynchronous Memory Bank Control 0 Register (Bits 15–0)

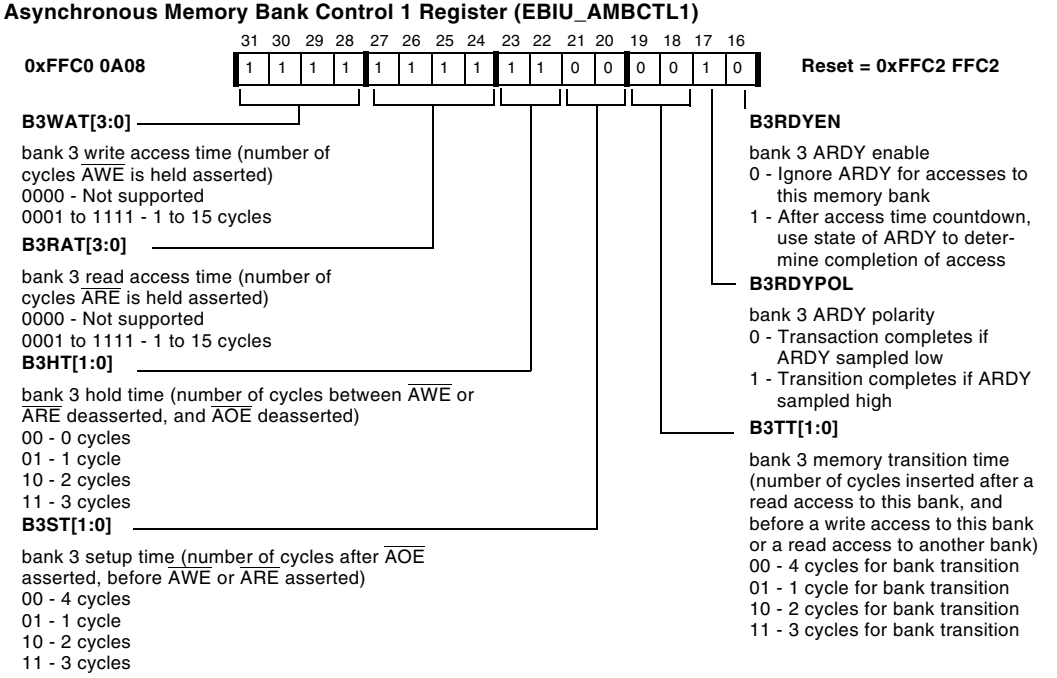


Figure 18-6. Asynchronous Memory Bank Control 1 Register (Bits 31–16)

Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)

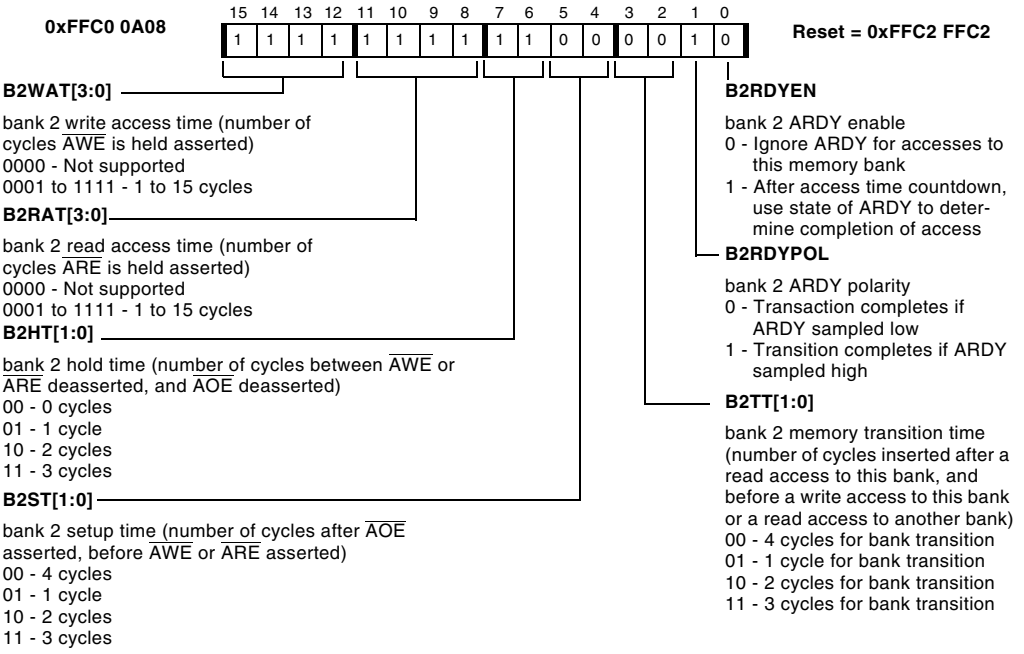


Figure 18-7. Asynchronous Memory Bank Control 1 Register (Bits 15–0)

Avoiding Bus Contention

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

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

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

ARDY Input Control

Each bank can be programmed to sample the ARDY input after the read or write access timer has counted down or to ignore this input signal. If enabled and disabled at the sample window, ARDY can be used to extend the access time as required. Note ARDY is synchronously sampled, therefore:

- Assertion and deassertion of ARDY to the processor must meet the data sheet setup and hold times. Failure to meet these synchronous specifications could result in meta-stable behavior internally. The processor's CLKOUT signal should be used to ensure synchronous transitions of ARDY.
- The ARDY pin must be stable (either asserted or deasserted) at the external interface on the cycle before the internal bank counter reaches 0; that is, more than one CLKOUT cycle before the scheduled rising edge of $\overline{\text{AWE}}$ or $\overline{\text{ARE}}$. This will determine whether the access is extended or not.
- Once the transaction has been extended as a result of ARDY being sampled in the "busy" state, the transaction will then complete in the cycle after ARDY is subsequently sampled in the "ready" state.

The polarity of ARDY is programmable on a per-bank basis. Since ARDY is not sampled until an access is in progress to a bank in which the ARDY enable is asserted, ARDY does not need to be driven by default. For more information, see "Adding Additional Wait States" on page 18-23.

Programmable Timing Characteristics

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

Asynchronous Accesses by Core Instructions

Some external memory accesses are caused by core instructions of the type:

```
R0.L = W[P0++] ; /* Read from external memory, where P0 points
to a location in external memory */
```

or:

```
W[P0++] = R0.L ; /* Write to external memory */
```

Asynchronous Reads

Figure 18-8 shows an asynchronous read bus cycle with timing programmed as setup = 2 cycles, read access = 2 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as follows.

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

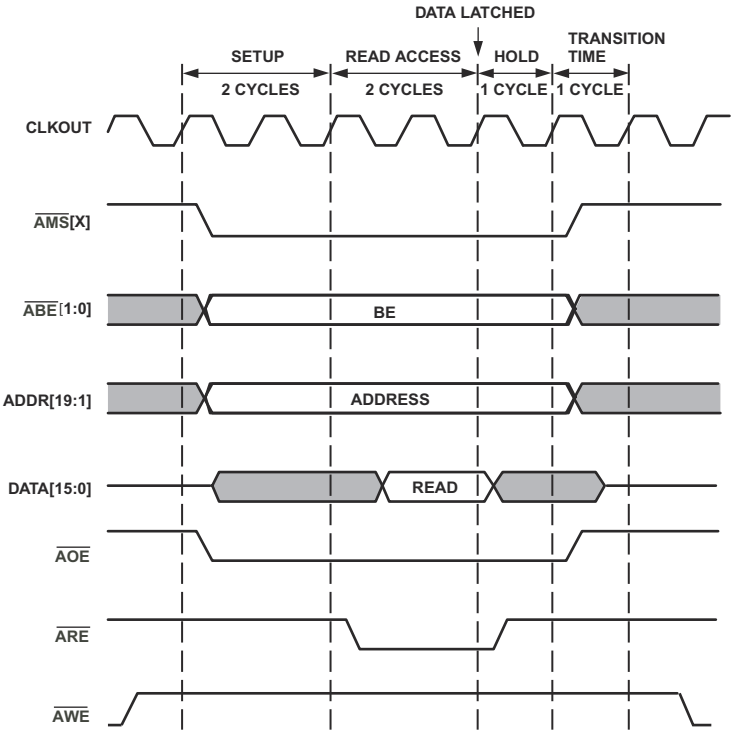


Figure 18-8. Asynchronous Read Bus Cycles

Read access is completed with the AMSx and AOE signals getting de-asserted. There are a few idle cycles before the next read operation starts. The number of idle cycles is a function of the CCLK/SCLK ratio. The number of idle cycles is 6 for a CCLK/SCLK ratio of 3, 4 for a CCLK/SCLK ratio of 5, and 3 for a CCLK/SCLK ratio of 10.

Asynchronous Writes

Figure 18-9 shows an asynchronous write bus cycle followed by an asynchronous read cycle to the same bank, with timing programmed as setup = 2 cycles, write access = 2 cycles, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous write bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, data buses, and $\overline{\text{ABE[1:0]}}$ become valid.
2. At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
3. At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.

Asynchronous read bus cycles proceed as follows.

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

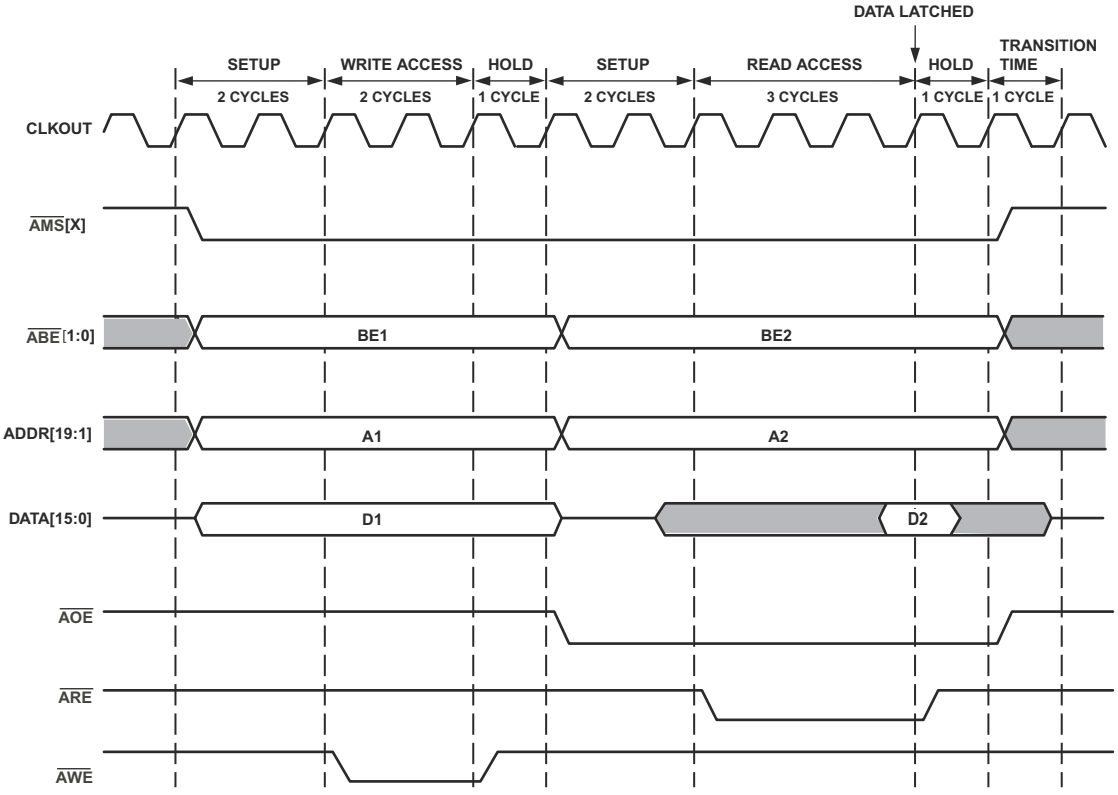


Figure 18-9. Asynchronous Write and Read Bus Cycles

Adding Additional Wait States

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

example of this behavior is shown in Figure 18-10, where setup = 2 cycles, read access = 4 cycles, and hold = 1 cycle. Note the read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

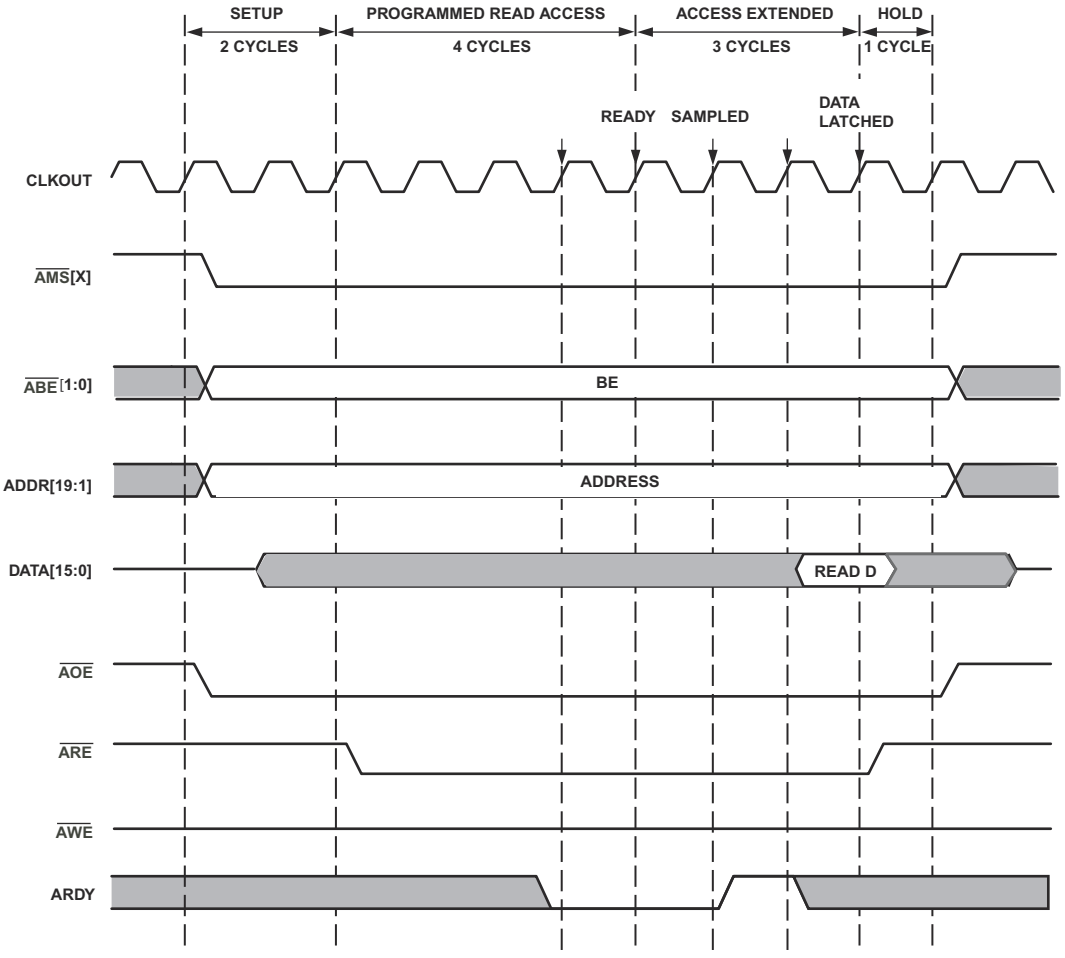


Figure 18-10. Inserting Wait States Using ARDY

Byte Enables

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

On-Chip Flash Memory

The ADSP-BF539F processors provide on-chip flash memory options. This flash memory is a separate die inside the package, and it can be mapped to any of the four asynchronous memory banks by connecting the $\overline{\texttt{FCE}}$ pin to the appropriate $\overline{\texttt{AMSx}}$ pin. If the $\overline{\texttt{FCE}}$ pin is connected to the $\overline{\texttt{AMS0}}$ pin, the processor will boot from the on-chip flash memory if configured to boot from parallel flash (BMODE[1:0] = b#01).

SDRAM Controller (SDC)

The SDRAM controller (SDC) enables the processor to transfer data to and from Synchronous DRAM (SDRAM) with a maximum frequency specified in *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*. The processor supports a glueless interface with one external bank of standard SDRAMs of 64 Mbit to 512 Mbit, with configurations x4, x8, and x16, up to a maximum total capacity of 128M bytes of SDRAM. This bank is controlled by the $\overline{\texttt{SMS}}$ memory select pin. The interface includes timing options to support additional buffers between the processor and SDRAM, to handle the capacitive loads of large memory arrays.

All inputs are sampled and all outputs are valid on the rising edge of the SDRAM clock output CLKOUT.

The EBIU SDC provides a glueless interface with standard SDRAMs. The SDRAM controller:

- Supports SDRAMs of 64M bit, 128M bit, 256M bit, and 512M bit with configurations of x4, x8, and x16
- Supports up to 128M byte of SDRAM in external SDRAM
- Supports SDRAM page sizes of 512 bytes, 1K byte, 2K byte, and 4K byte
- Supports four internal banks within the SDRAM
- Uses a programmable refresh counter to coordinate between varying clock frequencies and the refresh rate required by the SDRAM.
- Provides multiple timing options to support additional buffers between the processor and SDRAM
- Uses a separate pin (`SA10`) that enables the SDC to precharge SDRAM before issuing an Auto-Refresh or Self-Refresh command while the asynchronous memory controller has control of the EBIU port
- Supports self-refresh for standard SDRAMs and partial array self-refresh for mobile SDRAMs
- Provides two SDRAM power-up options
- Supports interleaved SDRAM bank accesses

Definition of Terms

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

Bank Activate command

The bank Activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the bank Activate command is issued to the SDRAM, the SDRAM opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The bank Activate command must be applied before a read or write command.

Burst Length

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command, respectively. The burst length is selected by writing certain bits to the SDRAM Mode register during the SDRAM power-up sequence.

Although the SDC supports only Burst Length = 1 mode, during a burst to SDRAM, the SDC applies the read or write command every cycle and keeps accessing the data. Therefore, the effective burst length is much greater than 1. In other words, setting Burst Length = 1 does not reduce the performance throughput.

Burst Stop Command

The Burst Stop command is one of several ways to terminate or interrupt a burst read or write operation.

Since the SDRAM burst length is always hardwired to be 1, the SDC does not support the Burst Stop command.

Burst Type

The burst type determines the address order in which the SDRAM delivers burst data after detecting a read command or stores burst data after detecting a write command. The burst type is programmed in the SDRAM during the SDRAM power-up sequence.

Since the SDRAM burst length is always programmed to be 1, the burst type does not matter. However, the SDC always sets the burst type to sequential-accesses-only during the SDRAM power-up sequence.

CAS Latency (CL)

The Column Address Strobe (CAS) latency is the delay in clock cycles between when the SDRAM detects the read command and when it provides the data at its output pins. The CAS latency is programmed in the SDRAM Mode register during the power-up sequence.

The speed grade of the SDRAM and the application's clock frequency determine the value of the CAS latency. The SDC can support CAS latency of two or three clock cycles. The selected CAS latency value must be programmed into the SDRAM memory global control register (`EBIU_SDGCTL`) before the SDRAM power-up sequence. See "EBIU_SDGCTL Register" on page 18-36.

CBR (CAS before RAS) Refresh or Auto-Refresh

When the SDC refresh counter times out, the SDC pre-charges all four banks of SDRAM and then issues an Auto-Refresh command to them. This causes the SDRAMs to generate an internal CBR refresh cycle. When the internal refresh completes, all four internal SDRAM banks are precharged.

DQM Pin Mask Function

The SDQM[1:0] pins provide a byte-masking capability on 8-bit writes to SDRAM. The DQM pins are used to block the input buffer of the SDRAM during partial write operations. The SDQM[1:0] pins are not used to mask data on partial read cycles. For write cycles, the data masks have a latency of zero cycles, permitting data writes when the corresponding SDQM[x]] pin is sampled low and blocking data writes when the SDQM[x] pin is sampled high on a byte-by-byte basis.

Internal Bank

There are several internal memory banks on a given SDRAM. The SDC supports interleaved accesses among the internal banks. The bank address can be thought of as part of the row address. The SDC assumes that all SDRAMs to which it interfaces have four internal banks and allows each activated bank to have a unique row address.

Mode Register

SDRAM devices contain an internal configuration register which allows specification of the SDRAM device's functionality. After power-up and before executing a read or write to the SDRAM memory space, the application must trigger the SDC to write to the SDRAM Mode register. The write to the SDRAM Mode register is triggered by writing a 1 to the PSSE bit in the SDRAM memory global control register (EBIU_SDGCTL) and then issuing a read or write transfer to the SDRAM address space. The initial read or write triggers the SDRAM power-up sequence to be run, which programs the SDRAM Mode register with the CAS latency from the EBIU_SDGCTL register. This initial read or write to SDRAM takes many cycles to complete.

Note for most applications, the SDRAM power-up sequence and writing of the Mode register needs to be done only once. Once the power-up sequence has completed, the PSSE bit should not be set again unless a change to the Mode register is desired. In this case, refer to "Managing SDRAM Refresh During PLL Transitions" on page 22-9.

Low power SDRAM devices may also contain an Extended Mode register. The EBIU enables programming of the Extended Mode register during power-up via the EMREN bit in the EBIU_SDGCTL register.

Page Size

Page size is the amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row. The page size can be calculated for 16-bit SDRAM banks with this formula:

- 16-bit SDRAM banks: page size = $2^{(CAW + 1)}$

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

Precharge Command

The Precharge command closes a specific internal bank in the active page or all internal banks in the page.

SDRAM Bank

The SDRAM bank is a region of memory that can be configured to 16M byte, 32M byte, 64M byte, or 128M byte and is selected by the $\overline{\text{SMS}}$ pin.

Do not confuse the "SDRAM internal banks" which are internal to the SDRAM and are selected with the bank address, with the "SDRAM bank" or "external bank" that is enabled by the $\overline{\text{SMS}}$ pin.

Self-Refresh

When the SDRAM is in Self-Refresh mode, the SDRAM internal timer initiates Auto-Refresh cycles periodically, without external control input. The SDC must issue a series of commands including the Self-Refresh command to put the SDRAM into this low power mode, and it must issue another series of commands to exit Self-Refresh mode. Entering Self-Refresh mode is programmable in the SDRAM memory global control register (EBIU_SDGCTL) and any access to the SDRAM address space causes the SDC to exit the SDRAM from Self-Refresh mode. See "Entering and Exiting Self-Refresh Mode (SRFS)" on page 18-41.

t_{RAS}

This is the required delay between issuing a bank Activate command and issuing a Precharge command, and between the Self-Refresh command and the exit from Self-Refresh. The TRAS bit field in the SDRAM memory global control register (EBIU_SDGCTL) is 4 bits wide and can be programmed to be 1 to 15 clock cycles long. "Selecting the Bank Activate Command Delay (TRAS)" on page 18-44.

t_{RC}

This is the required delay between issuing successive bank Activate commands to the same SDRAM internal bank. This delay is not directly programmable. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

t_{RCD}

This is the required delay between a bank Activate command and the start of the first Read or Write command. The TRCD bit field in the SDRAM memory global control register (EBIU_SDGCTL) is three bits wide and can be programmed to be from 1 to 7 clock cycles long.

t_{RFC}

This is the required delay between issuing an Auto-Refresh command and a bank Activate command and between issuing successive Auto-Refresh commands. This delay is not directly programmable and is assumed to be equal to t_{RC}. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

t_{RP}

This is the required delay between issuing a Precharge command and:

- issuing a bank Activate command
- issuing an Auto-Refresh command
- issuing a Self-Refresh command

The TRP bit field in the SDRAM memory global control register (EBIU_SDGCTL) is three bits wide and can be programmed to be 1 to 7 clock cycles long. "Selecting the Precharge Delay (TRP)" on page 18-46.

t_{RRD}

This is the required delay between issuing a bank A Activate command and a bank B Activate command. This delay is not directly programmable and is assumed to be $t_{RCD} + 1$.

t_{WR}

This is the required delay between a Write command (driving write data) and a Precharge command. The TWR bit field in the SDRAM memory global control register (EBIU_SDGCTL) is two bits wide and can be programmed to be from 1 to 3 clock cycles long.

t_{XSR}

This is the required delay between exiting Self-Refresh mode and issuing the Auto-Refresh command. This delay is not directly programmable and is assumed to be equal to t_{RC}. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

SDRAM Configurations Supported

The following table shows all possible bank sizes, bank widths and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC.

Table 18-4. SDRAM Discrete Component Configurations Supported

Bank Size (M byte)	Bank Width (Bits)	SDRAM		
		Size (M bit)	Configuration	Number of Chips
16	16	32	2M x 4 x 4 banks	4
16	16	64	2M x 8 x 4 banks	2
16	16	128	2M x 16 x 4 banks	1
32	16	64	4M x 4 x 4 banks	4
32	16	128	4M x 8 x 4 banks	2
32	16	256	4M x 16 x 4 banks	1
64	16	128	8M x 4 x 4 banks	4
64	16	256	8M x 8 x 4 banks	2
64	16	512	8M x 16 x 4 banks	1
128	16	256	16M x 4 x 4 banks	4
128	16	512	16M x 8 x 4 banks	2
128	16	1024	16M x 16 x 4 banks	1

Example SDRAM System Block Diagrams

Figure 18-11 shows a block diagram of the SDRAM interface. In this example, the SDRAM interface connects to two 64 Mbit (x8) SDRAM devices to form one external bank of 16M bytes of memory. The same address and control bus feeds both SDRAM devices.

The SDC includes a separate address pin (SA10) to enable the execution of Auto-Refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a Precharge command to the SDRAM before it issues an Auto-Refresh command.

In addition, the SA10 pin allows the SDC to enter and exit Self-Refresh mode in parallel with any asynchronous memory access. The SA10 pin (instead of the ADDR[11] pin) should be directly connected to the SDRAM A10 pin. During the Precharge command, SA10 is used to indicate that a Precharge All should be done. During a bank Activate command, SA10 outputs the internal row address bit, which should be multiplexed to the A10 SDRAM input. During Read and Write commands, SA10 is used to disable the auto-precharge function of SDRAMs.

SDRAM systems do not use the ADDR[11] pin.

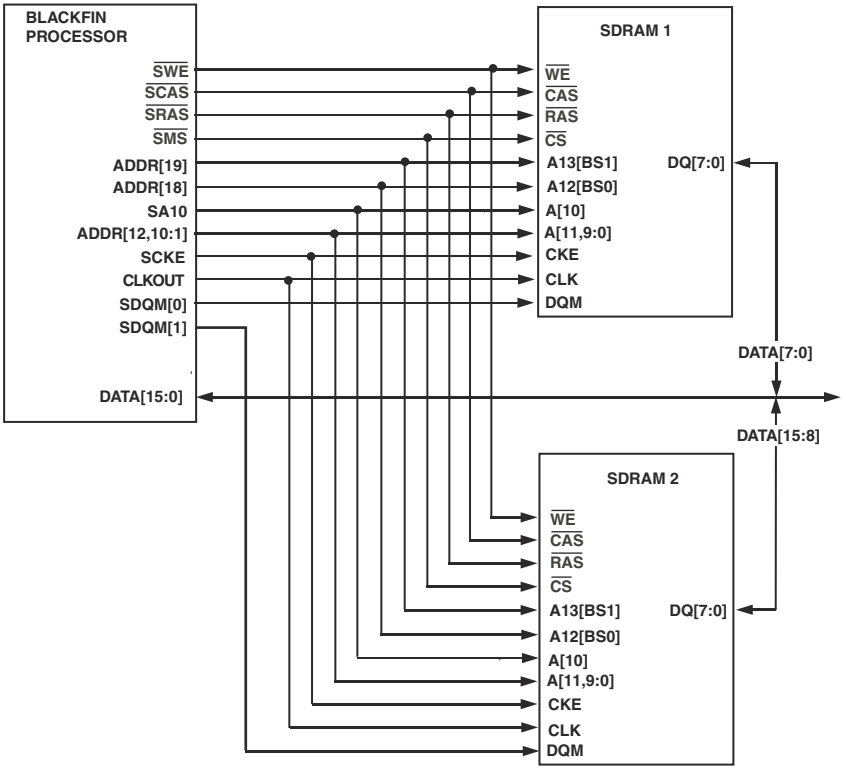


Figure 18-11. 16M byte SDRAM System Example

Executing a Parallel Refresh Command

The SDC includes a separate address pin (SA10) to enable the execution of Auto-Refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a Precharge command to the SDRAM before it issues an Auto-Refresh command. In addition, the SA10 pin allows the SDC to enter and exit Self-Refresh mode in parallel with any asynchronous memory access.

The SA10 pin should be directly connected to the A10 pin of the SDRAM (instead of to the ADDR[10] pin). During the Precharge command, SA10 is used to indicate that a Precharge All should be done. During a bank Activate command, SA10 outputs the internal row address bit, which should be multiplexed to the A10 SDRAM input. During Read and Write commands, SA10 is used to disable the auto-precharge function of SDRAMs.

EBIU_SDGCTL Register

The SDRAM memory global control register (EBIU_SDGCTL) includes all programmable parameters associated with the SDRAM access timing and configuration. Figure 18-12 shows the EBIU_SDGCTL register bit definitions.

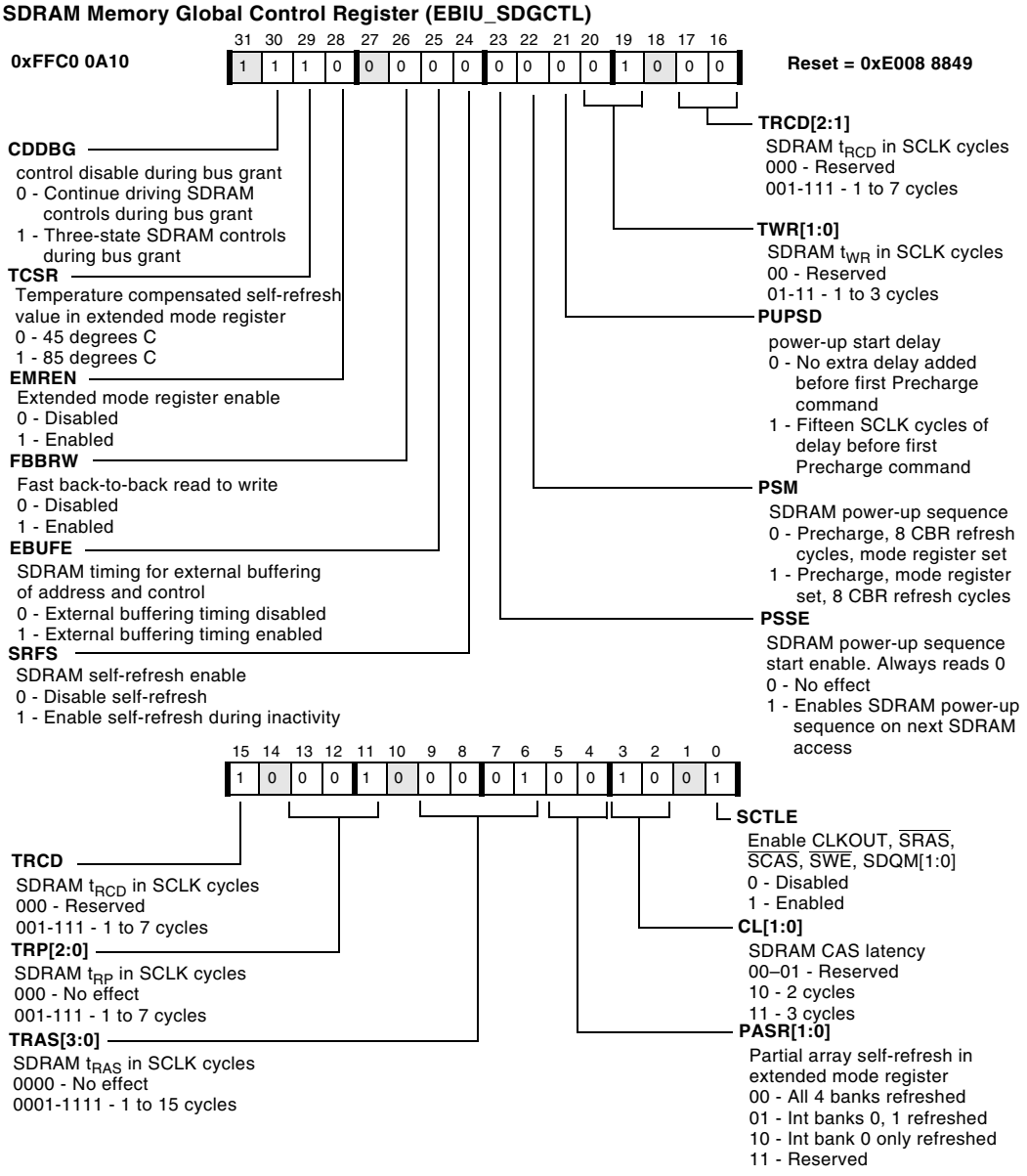


Figure 18-12. SDRAM Memory Global Control Register

The SCTLE bit is used to enable or disable the SDC. If SCTLE is disabled, any access to SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 18-8. When SCTLE is disabled, all SDC control pins are in their inactive states and the SDRAM clock is not running. The SCTLE bit must be enabled for SDC operation and is enabled by default at reset.

The CAS Latency (CL), SDRAM t_{RAS} Timing (TRAS), SDRAM t_{RP} Timing (TRP), SDRAM t_{RCD} Timing (TRCD), and SDRAM t_{WR} Timing (TWR) bits should be programmed based on the system clock frequency and the timing specifications of the SDRAM used.

The user must ensure that `tRAS + tRP >= max(tRC,tRFC,tXSR)`.

The PSM and PSSE bits work together to specify and trigger an SDRAM power-up (initialization) sequence. If the PSM bit is set to 1, the SDC does a Precharge All command, followed by a Load Mode register command, and then does eight Auto-Refresh cycles. If the PSM bit is cleared, the SDC does a Precharge All command, followed by eight Auto-Refresh cycles, and then a Load Mode register command. Two events must occur before the SDC does the SDRAM power-up sequence:

- The PSSE bit must be set to 1 to enable the SDRAM power-up sequence.
- A read or write access must be done to enabled SDRAM address space in order to have the external bus granted to the SDC so that the SDRAM power-up sequence may occur.

The SDRAM power-up sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM power-up sequence. Note there is a latency for this first access to SDRAM because the SDRAM power-up sequence takes many cycles to complete.

Before executing the SDC power-up sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as specified by the SDRAM specification.

The Power-up Start Delay bit (PUPSD) optionally delays the power-up start sequence for 15 SCLK cycles. This is useful for multiprocessor systems sharing an external SDRAM. If the bus has been previously granted to the other processor before power-up and Self-Refresh mode is used when switching bus ownership, then the PUPSD bit can be used to guarantee a sufficient period of inactivity from self-refresh to the first Precharge command in the power-up sequence in order to meet the exit self-refresh time (t_{XSR}) of the SDRAM.

When the SRFS bit is set to 1, Self-Refresh mode is triggered. Once the SDC completes any active transfers, the SDC executes the sequence of commands to put the SDRAM into Self-Refresh mode. The next access to an enabled SDRAM bank causes the SDC to execute the commands to exit the SDRAM from Self-Refresh and execute the access. See "Entering and Exiting Self-Refresh Mode (SRFS)" on page 18-41 for more information about the SRFS bit.

The EBUFE bit is used to enable or disable external buffer timing. When buffered SDRAM modules or discrete register-buffers are used to drive the SDRAM control inputs, EBUFE should be set to 1. Using this setting adds a cycle of data buffering to read and write accesses. See "Setting the SDRAM Buffering Timing Option (EBUFE)" on page 18-42 for more information about the EBUFE bit.

The FBBRW bit enables an SDRAM read followed by write to occur on consecutive cycles. In many systems, this is not possible because the turn-off time of the SDRAM data pins is too long, leading to bus contention with the succeeding write from the processor. When this bit is 0, a clock cycle is inserted between read accesses followed immediately by write accesses.

The EMREN bit enables programming of the Extended Mode register during startup. The Extended Mode register is used to control SDRAM power consumption in certain mobile low power SDRAMs. If the EMREN bit is enabled, then the TCSR and PASR[1:0] bits control the value written to the Extended Mode register. The PASR bits determine how many SDRAM internal banks are refreshed during Self-Refresh. The TCSR bit signals to the SDRAM the worst case temperature range for the system, and thus how often the SDRAM internal banks need to be refreshed during Self-Refresh.

The CDDBG bit is used to enable or disable the SDRAM control signals when the external memory interface is granted to an external controller. If this bit is set to a 1, then the control signals are three-stated when bus grant is active. Otherwise, these signals continue to be driven during grant. If the bit is set and the external bus is granted, all SDRAM internal banks are assumed to have been changed by the external controller. This means a precharge is required on each bank prior to use after control of the external bus is re-established. The control signals affected by this pin are $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$, $\overline{\text{SMS}}$, SA10, SCKE, and CLKOUT.

All reserved bits in this register must always be written with 0s.

Setting the SDRAM Clock Enable (SCTLE)

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

- SCTLE = 0

  Disable all SDRAM control pins (control pins negated, CLKOUT low)

- SCTLE = 1

  Enable all SDRAM control pins (CLKOUT toggles)

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

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

If an access occurs to the SDRAM address space while SCTLE is 0, the access generates an internal bus error and the access does not occur externally. For more information, see "Error Detection" on page 18-8. With careful software control, the SCTLE bit can be used in conjunction with Self-Refresh mode to further lower power consumption. However, SCTLE must remain enabled at all times when the SDC is needed to generate Auto-Refresh commands to SDRAM.

Entering and Exiting Self-Refresh Mode (SRFS)

The SDC supports SDRAM Self-Refresh mode. In Self-Refresh mode, the SDRAM performs refresh operations internally—without external control—reducing the SDRAM power consumption.

The SRFS bit in EBIU_SDGCTL enables the start of Self-Refresh mode:

- SRFS = 0

  Disable Self-Refresh mode

- SRFS = 1

  Enable Self-Refresh mode

When SRFS is set to 1, once the SDC enters an idle state it issues a Precharge command if necessary, and then issues a Self-Refresh command. If an internal access is pending, the SDC delays issuing the Self-Refresh command until it completes the pending SDRAM access and any subsequent pending access requests. Refer to "SDC Commands" on page 18-59 for more information.

Once the SDRAM device enters into Self-Refresh mode, the SDRAM controller asserts the SDSRA bit in the SDRAM control status register (EBIU_SDSTAT).

The SDRAM device exits Self-Refresh mode only when the SDC receives a core or DMA access request. In conjunction with the SRFS bit, 2 possibilities are given to exit the self-refresh mode:

- If SRFS bit is set before the request, the SDC exits self-refresh and remains in auto-refresh mode.
- If SRFS bit is cleared before the request, the SDC exits self-refresh only for a single request and returns back to self-refresh mode until a new request is coming.

Note once the SRFS bit is set to 1, the SDC enters Self-Refresh mode when it finishes pending accesses. There is no way to cancel the entry to Self-Refresh mode.

Setting the SDRAM Buffering Timing Option (EBUFE)

To meet overall system timing requirements, systems that employ several SDRAM devices connected in parallel may require buffering between the processor and multiple SDRAM devices. This buffering generally consists of a register and driver.

To meet such timing requirements and to allow intermediary registration, the SDC supports pipelining of SDRAM address and control signals.

The EBUFE bit in the EBIU_SDGCTL register enables this mode:

- EBUFE = 0

  Disable external buffering timing

- EBUFE = 1

  Enable external buffering timing

When EBUFE = 1, the SDRAM controller delays the data in write accesses by one cycle, enabling external buffer registers to latch the address and controls. In read accesses, the SDRAM controller samples data one cycle later to account for the one-cycle delay added by the external buffer registers. When external buffering timing is enabled, the latency of all accesses is increased by one cycle.

Selecting the CAS Latency Value (CL)

The CAS latency value defines the delay, in number of clock cycles, between the time the SDRAM detects the Read command and the time it provides the data at its output pins.

CAS latency does not apply to write cycles.

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

- CL = b#00

  Reserved

- CL = b#01

  Reserved

- CL = b#10

  2 clock cycles

- CL = b#11

  3 clock cycles

Generally, the frequency of operation determines the value of the CAS latency. For specific information about setting this value, consult the SDRAM device documentation.

Selecting the Bank Activate Command Delay (TRAS)

The t_{RAS} value (bank Activate command delay) defines the required delay, in number of clock cycles, between the time the SDC issues a bank Activate command and the time it issues a Precharge command. The SDRAM must also remain in Self-Refresh mode for at least the time period specified by t_{RAS}. The t_{RP} and t_{RAS} values define the t_{RFC}, t_{RC}, and t_{XSR} values. See page -31 for more information.

The t_{RAS} parameter allows the processor to adapt to the timing requirements of the system's SDRAM devices.

The TRAS bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RAS} value. Any value between 1 and 15 clock cycles can be selected. For example:

- TRAS = b#0000

  No effect

- TRAS = b#0001

  1 clock cycle

- TRAS = b#0010

  2 clock cycles

- TRAS = b#1111

  15 clock cycles

For specific information on setting this value, consult the SDRAM device documentation.

Selecting the RAS to CAS Delay (TRCD)

The t_{RCD} value (RAS to CAS delay) defines the delay for the first read or write command after a row activate command, in number of clock cycles. The t_{RCD} parameter allows the processor to adapt to the timing requirements of the system's SDRAM devices.

The t_{RCD} bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RCD} value. Any value between 1 and 7 clock cycles may be selected. For example:

- TRCD = reserved

  No effect

- TRCD = b#001

  1 clock cycle

- TRCD = b#010

  2 clock cycles

- TRCD = b#111

  7 clock cycles

Selecting the Precharge Delay (TRP)

The t_{RP} value (Precharge delay) defines the required delay, in number of clock cycles, between the time the SDC issues a Precharge command and the time it issues a bank Activate command. The t_{RP} also specifies the time required between Precharge and Auto-Refresh, and between Precharge and Self-Refresh. The t_{RP} and t_{RAS} values define the t_{RFC}, t_{RC}, and t_{XSR} values.

This parameter enables the application to accommodate the SDRAM timing requirements.

The TRP bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RP} value. Any value between 1 and 7 clock cycles may be selected. For example:

- TRP = b#000

  No effect

- TRP = b#001

  1 clock cycle

- TRP = b#010

  2 clock cycles

- TRP = b#111

  7 clock cycles

Selecting the Write to Precharge Delay (TWR)

The t_{WR} value defines the required delay, in number of clock cycles, between the time the SDC issues a Write command (drives write data) and a Precharge command.

This parameter enables the application to accommodate the SDRAM timing requirements.

The TWR bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{WR} value. Any value between 1 and 3 clock cycles may be selected. For example:

- TWR = b#00

  Reserved

- TWR = b#01

  1 clock cycle

- TWR = b#10

  2 clock cycles

- TWR = b#11

  3 clock cycles

EBIU_SDBCTL Register

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

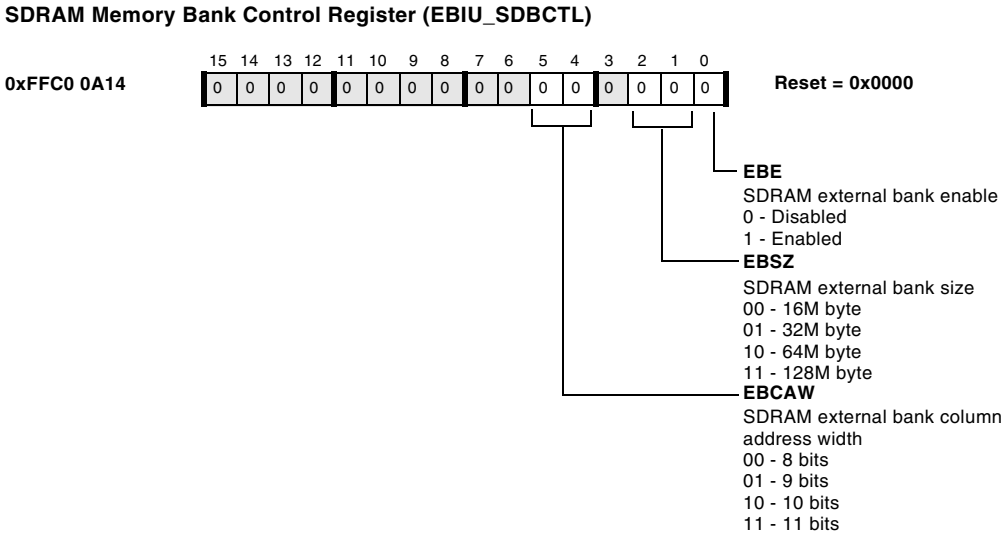


Figure 18-13. SDRAM Memory Bank Control Register

The EBIU_SDBCTL register stores the configuration information for the SDRAM bank interface. The EBIU supports 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit SDRAM devices with x4, x8, x16 configurations. Table 18-4 on page 18-33 maps SDRAM density and I/O width to the supported EBSZ encodings. See "SDRAM External Memory Size" on page 18-54 for more information on bank starting address decodes.

The SDC determines the internal SDRAM page size from the EBCAW parameters. Page sizes of 512 B, 1K byte, 2K byte, and 4K byte are supported. Table 18-5 on page 18-50 shows the page size and breakdown of the internal address (IA[31:0], as seen from the core or DMA) into the row, bank, column, and byte address. The column address and the byte address together make up the address inside the page.

The EBE bit in the EBIU_SDBCTL register is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 18-8.

Table 18-5. Internal Address Mapping

Bank Width (bits)	Bank Size (Mbyte)	Col. Addr. Width (CAW)	Page Size (K Byte)	Bank Address	Row Address	Page	
						Column Address	Byte Address
16	128	11	4	IA[26:25]	IA[24:12]	A[11:1]	IA[0]
16	128	10	2	IA[26:25]	IA[24:11]	IA[10:1]	IA[0]
16	128	9	1	1A[26:25]	IA[24:10]	IA[9:1]	IA[0]
16	128	8	.5	IA[26:25]	IA[24:9]	IA[8:1]	IA[0]
16	64	11	4	IA[25:24]	IA[23:12]	IA[11:1]	IA[0]
16	64	10	2	IA[25:24]	IA[23:11]	IA[10:1]	IA[0]
16	64	9	1	IA[25:24]	IA[23:10]	IA[9:1]	IA[0]
16	64	8	.5	IA[25:24]	IA[23:9]	IA[8:1]	IA[0]
16	32	11	4	IA[24:23]	IA[22:12]	IA[11:1]	IA[0]
16	32	10	2	IA[24:23]	IA[22:11]	IA[10:1]	IA[0]
16	32	9	1	IA[24:23]	IA[22:10]	IA[9:1]	IA[0]
16	32	8	.5	IA[24:23]	IA[22:9]	IA[8:1]	IA[0]
16	16	11	4	IA[23:22]	IA[21:12]	IA[11:1]	IA[0]
16	16	10	2	IA[23:22]	IA[21:11]	IA[10:1]	IA[0]
16	16	9	1	IA[23:22]	IA[21:10]	IA[9:1]	IA[0]
16	16	8	.5	IA[23:22]	IA[21:9]	IA[8:1]	IA[0]

For information on how to connect to SDRAMs smaller than 16M byte, see "Using SDRAMs Smaller than 16M byte" on page 22-7.

EBIU_SDSTAT Register

The SDRAM control status register (EBIU_SDSTAT) provides information on the state of the SDC. This information can be used to determine when it is safe to alter SDC control parameters or it can be used as a debug aid. The SDEASE bit of this register is sticky. Once it has been set, software must explicitly write a 1 to the bit to clear it. Writes have no effect on the other status bits, which are updated by the SDC only. This SDC MMR is 16 bits wide.

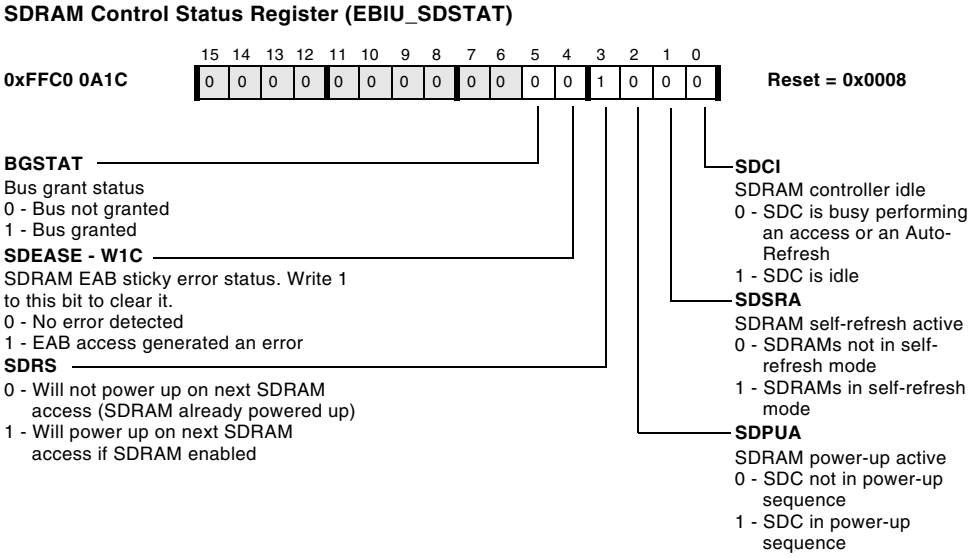


Figure 18-14. SDRAM Control Status Register

EBIU_SDRRC Register

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

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

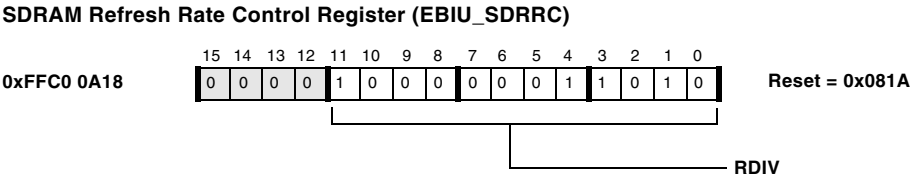


Figure 18-15. SDRAM Refresh Rate Control Register

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

RDIV = ((fSCLK ´ tREF) / NRA) – (tRAS + tRP)

Where:

- f_{SCLK} = SDRAM clock frequency (system clock frequency)
- t_{REF} = SDRAM refresh period
- NRA = Number of row addresses in SDRAM (refresh cycles to refresh whole SDRAM)

- t_{RAS} = Active to Precharge time (TRAS in the SDRAM memory global control register) in number of clock cycles
- t_{RP} = RAS to Precharge time (TRP in the SDRAM memory global control register) in number of clock cycles

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

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

- f_{SCLK} = 133 MHz
- t_{REF} = 64 ms
- NRA = 4096 row addresses
- t_{RAS} = 2
- t_{RP} = 2

The equation for RDIV yields:

RDIV = ((133 x 106 x 64 x 10-3) / 4096) – (2 + 2) = 2074 clock cycles

This means RDIV is 0x81A (hexadecimal equivalent for 2074) and the SDRAM Refresh Rate control register should be written with 0x081A.

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

Refer to "Managing SDRAM Refresh During PLL Transitions" on page 22-9 for a detailed discussion of the process for changing the PLL frequency when using SDRAM.

SDRAM External Memory Size

The total amount of external SDRAM memory addressed by the processor is controlled by the EBSZ bits of the EBIU_SDBCTL register (see Table 18-6). Accesses above the range shown for a specialized EBSZ value results in an internal bus error and the access does not occur. For more information, see "Error Detection" on page 18-8.

Table 18-6. Bank Size Encodings

EBSZ	Bank Size (Mbyte)	Valid SDRAM Addresses
b#00	16	0x0000 0000 – 0x00FF FFFF
b#01	32	0x0000 0000 – 0x01FF FFFF
b#10	64	0x0000 0000 – 0x03FF FFFF
b#11	128	0x0000 0000 – 0x07FF FFFF

SDRAM Address Mapping

To access SDRAM, the SDC multiplexes the internal 32-bit non-multiplexed address into a row address, a column address, a bank address, and the byte data masks for the SDRAM device. See Figure 18-16. The lowest bit is mapped to byte data masks, the next bits are mapped into the column address, the next bits are mapped into the row address, and the final two bits are mapped into the bank address. This mapping is based on the EBSZ and EBCAW parameters programmed into the SDRAM memory bank control register.

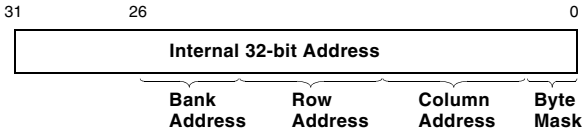


Figure 18-16. Multiplexed SDRAM Addressing Scheme

16-Bit Wide SDRAM Address Muxing

The following table shows the connection of the address pins with the SDRAM device pins.

Table 18-7. SDRAM Address Connections for 16-bit Banks

External Address Pin	SDRAM Address Pin
ADDR[19]	BA[1]
ADDR[18]	BA[0]
ADDR[16]	A[15]
ADDR[15]	A[14]
ADDR[14]	A[13]
ADDR[13]	A[12]
ADDR[12]	A[11]
SA[10]	A[10]
ADDR[10]	A[9]
ADDR[9]	A[8]
ADDR[8]	A[7]
ADDR[7]	A[6]
ADDR[6]	A[5]
ADDR[5]	A[4]
ADDR[4]	A[3]

Table 18-7. SDRAM Address Connections for 16-bit Banks (Cont'd)

External Address Pin	SDRAM Address Pin
ADDR[3]	A[2]
ADDR[2]	A[1]
ADDR[1]	A[0]

Data Mask (SDQM[1:0]) Encodings

During write transfers to SDRAM, the SDQM[1:0] pins are used to mask writes to bytes that are not accessed. Table 18-8 on page 18-56 shows the SDQM[1:0] encodings for 16-bit wide SDRAM based on the internal transfer address bit IA[0] and the transfer size.

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

The only time that the SDQM[1:0] pins are high is when bytes are masked during write transfers to the SDRAM. At all other times, the SDQM[1:0] pins are held low.

Table 18-8. SDQM[1:0] Encodings During Writes

Internal Address IA[0]	Internal Transfer Size		
	byte	2 bytes	4 bytes
0	SDQM[1] = 1 SDQM[0] = 0	SDQM[1] = 0 SDQM[0] = 0	SDQM[1] = 0 SDQM[0] = 0
1	SDQM[1] = 0 SDQM[0] = 1		

SDC Operation

The SDC uses a burst length = 1 for read and write operations. Whenever a page miss occurs, the SDC executes a Precharge command followed by a bank Activate command before executing the Read or Write command. If there is a page hit, the Read or Write command can be given immediately without requiring the Precharge command.

For SDRAM Read commands, there is a latency from the start of the Read command to the availability of data from the SDRAM, equal to the CAS latency. This latency is always present for any single read transfer. Subsequent reads do not have latency.

A programmable refresh counter is provided. It can be programmed to generate background Auto-Refresh cycles at the required refresh rate based on the clock frequency used. The refresh counter period is specified with the RDIV field in the SDRAM Refresh Rate control register.

To allow Auto-Refresh commands to execute in parallel with any AMC access, a separate A10 pin (SA10) is provided. All the SDRAM internal banks are precharged before issuing an Auto-Refresh command.

The internal 32-bit non-multiplexed address is multiplexed into a row address, a column address, a bank select address, and data masks. Bit0 for 16-bit wide SDRAMs is used to generate the data masks. The next lowest bits are mapped into the column address, next bits are mapped into the row address, and the final two bits are mapped into the internal bank address. This mapping is based on the EBCAW and EBSZ values programmed into the SDRAM memory bank control register.

SDC Configuration

After a processor's hardware or software reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the power-up sequence, ensure the clock to the SDRAM is enabled after the power has stabilized for the proper

amount of time (as specified by the SDRAM). In order to set up the SDC and start the SDRAM power-up sequence for the SDRAMs, the SDRAM Refresh Rate control register (EBIU_SDRRC), the SDRAM memory bank control register (EBIU_SDBCTL), and SDRAM memory global control register (EBIU_SDGCTL) must be written, and a transfer must be started to SDRAM address space. The SDRS bit of the SDRAM control status register can be checked to determine the current state of the SDC. If this bit is set, the SDRAM power-up sequence has not been initiated.

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

The EBIU_SDBCTL register should be written to describe the sizes and SDRAM memory configuration used (EBSZ and EBCAW) and to enable the external bank (EBE). Note until the SDRAM power-up sequence has been started, any access to SDRAM address space, regardless of the state of the EBE bit, generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 18-8. After the SDRAM power-up sequence has completed, if the external bank is disabled, any transfer to it results in a hardware error interrupt, and the SDRAM transfer does not occur.

The EBIU_SDGCTL register is written:

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

Note if SCTLE is disabled, any access to SDRAM address space generates an internal bus error and the access does not occur externally. For more information, see "Error Detection" on page 18-8.

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

SDC Commands

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

- Precharge All: Precharges all banks
- Single Precharge: Precharges a single bank
- Bank Activate: Activates a page in the required SDRAM internal bank
- Load Mode register: Initializes the SDRAM operation parameters during the power-up sequence
- Load Extended Mode register: Initializes mobile SDRAM operation parameters during the power-up sequence
- Read/Write
- Auto-Refresh: Causes the SDRAM to execute a CAS before RAS refresh

- Self-Refresh: Places the SDRAM in self-refresh mode, in which the SDRAM powers down and controls its refresh operations internally
- NOP/Command Inhibit: No operation

The following table shows the SDRAM pin state during SDC commands.

Table 18-9. Pin State During SDC Commands

Command	$\overline{SMS}$	$\overline{SCAS}$	$\overline{SRAS}$	$\overline{SWE}$	SCKE	SA10
Precharge All	low	high	low	low	high	high
Single Precharge	low	high	low	low	high	low
bank Activate	low	high	low	high	high	
Load Mode register	low	low	low	low	high	
Load Extended Mode register	low	low	low	low	high	low
Read	low	low	high	high	high	low
Write	low	low	high	low	high	low
Auto-Refresh	low	low	low	high	high	
Self-Refresh	low	low	low	high	low	
NOP	low	high	high	high	high	
Command Inhibit	high	high	high	high	high	

Precharge Commands

The Precharge All command is given to precharge all internal banks at the same time before executing an auto-refresh. For a page miss during reads or writes in a specific internal SDRAM bank, the SDC uses the Single Precharge command to that bank.

Bank Activate Command

The bank Activate command is required if the next data access is in a different page. The SDC executes the Precharge command, followed by a bank Activate command, to activate the page in the desired SDRAM internal bank.

The SDC supports bank interleaving (opening up to 4 internal SDRAM banks at a time). This results in an effective size of 4 pages. The address mapping indicates the start address of each internal bank.

Bank interleaving is accomplished by switching between 4 internal SDRAM banks without any stalls between the pages.

Load Mode Register Command

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

The Load Mode register command initializes these parameters:

- Burst length = 1, bits 2–0, always 0
- Wrap type = sequential, bit 3, always 0
- Ltmode = latency mode (CAS latency), bits 6–4, programmable in the `EBIU_SDGCTL` register
- Bits 14–7, always 0

While executing the Load Mode register command, the unused address pins are set to 0. During the two clock cycles following the Load Mode register command, the SDC issues only NOP commands.

For low power mobile SDRAMs that include an Extended Mode register, this register is programmed during power-up sequence if the EMREN bit is set in the EBIU_SDGCTL register.

The Extended Mode register is initialized with these parameters:

- Partial Array Self-Refresh, bits 2–0, bit 2 always 0, bits 1–0 programmable in EBIU_SDGCTL
- Temperature Compensated Self-Refresh, bits 4–3, bit 3 always 1, bit 4 programmable in EBIU_SDGCTL
- Bits 12–5, always 0, and bit 13 always 1

Read/Write Command

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

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

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

Auto-Refresh Command

The SDRAM internally increments the refresh address counter and causes a CAS before RAS (CBR) refresh to occur internally for that address when the Auto-Refresh command is given. The SDC generates an Auto-Refresh command after the SDC refresh counter times out. The RDIV value in the SDRAM Refresh Rate control register must be set so that all addresses are refreshed within the t_{REF} period specified in the SDRAM timing specifications. This command is issued to the external bank whether or not it is enabled (EBE in the SDRAM memory global control register). Before executing the Auto-Refresh command, the SDC executes a Precharge All command to the external bank. The next Activate command is not given until the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) is met.

Auto-Refresh commands are also issued by the SDC as part of the power-up sequence and also after exiting Self-Refresh mode.

Self-Refresh Command

The Self-Refresh command causes refresh operations to be performed internally by the SDRAM, without any external control. This means that the SDC does not generate any Auto-Refresh cycles while the SDRAM is in Self-Refresh mode. Before executing the Self-Refresh command, all internal banks are precharged. Self-Refresh mode is enabled by writing a 1 to the SRFS bit of the SDRAM memory global control register (EBIU_SDGCTL). After issuing the Self-Refresh command, the SDC drives SCKE low. This puts the SDRAM into a power down mode (SCKE = 0, $\overline{\text{SRAS}}/\overline{\text{SMS}}/\overline{\text{SCAS}}/\overline{\text{SWE}}$ = 1) Before exiting Self-Refresh mode, the SDC asserts SCKE. The SDRAM remains in Self-Refresh mode for at least t_{RAS} and until an internal access to SDRAM space occurs. When an internal access occurs causing the SDC to exit the SDRAM from Self-Refresh mode, the SDC waits to meet the t_{XSR} specification ($t_{XSR} = t_{RAS} + t_{RP}$) and then issues an Auto-Refresh command. After the Auto-Refresh command, the SDC waits for the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) to be met before executing the Activate command for the transfer that caused the SDRAM

to exit Self-Refresh mode. Therefore, the latency from when a transfer is received by the SDC while in Self-Refresh mode, until the Activate command occurs for that transfer, is 2 x (t_{RAS} + t_{RP}).

Note CLKOUT is not disabled by the SDC during Self-Refresh mode. However, software may disable the clock by clearing the SCTLE bit in EBIU_SDGCTL. The application software should ensure that all applicable clock timing specifications are met before the transfer to SDRAM address space which causes the controller to exit Self-Refresh mode. If a transfer occurs to SDRAM address space when the SCTLE bit is cleared, an internal bus error is generated, and the access does not occur externally, leaving the SDRAM in Self-Refresh mode. For more information, see "Error Detection" on page 18-8.

No Operation/Command Inhibit Commands

The No Operation (NOP) command to the SDRAM has no effect on operations currently in progress. The Command Inhibit command is the same as a NOP command; however, the SDRAM is not chip-selected. When the SDC is actively accessing the SDRAM but needs to insert additional commands with no effect, the NOP command is given. When the SDC is not accessing the SDRAM, the Command Inhibit command is given.

SDRAM Timing Specifications

To support key timing requirements and power-up sequences for different SDRAM vendors, the SDC provides programmability for t_{RAS}, t_{RP}, t_{RCD}, t_{WR}, and the power-up sequence mode (see "EBIU_SDGCTL Register" on page 18-36). CAS latency should be programmed in the EBIU_SDGCTL register based on the frequency of operation. (Refer to the SDRAM vendor's data sheet for more information.)

For other parameters, the SDC assumes:

- Bank Cycle Time: $t_{RC} = t_{RAS} + t_{RP}$
- Refresh Cycle Time: $t_{RFC} = t_{RAS} + t_{RP}$
- Exit Self-Refresh Time: $t_{XSR} = t_{RAS} + t_{RP}$
- Load Mode register to Activate Time: t_{MRD} or t_{RSC} = 3 clock cycles
- Page-Miss Penalty = $t_{RP} + t_{RCD}$
- Row (bank A) to Row (bank B) Active Time: $t_{RRD} = t_{RCD} + 1$

SDRAM Performance

Table 7-3 on page 7-12 lists the data throughput rates for the core or DMA read/write accesses to 16-bit wide SDRAM. For this example, assume all cycles are SCLK cycles and the following SCLK frequency and SDRAM parameters are used:

- SCLK frequency = 133 MHz
- CAS latency = 2 cycles (CL = 2)
- No SDRAM buffering (EBUFE = 0)
- RAS precharge (t_{RP}) = 2 cycles (TRP = 2)
- RAS to CAS delay (t_{RCD}) = 2 cycles (TRCD = 2)
- Active command time (t_{RAS}) = 5 cycles (TRAS = 5)

When the external buffer timing (EBUFE = 1 in the SDRAM memory global control register) and/or CAS latency of 3 (CL = b#11 in the SDRAM memory global control register) is used, all accesses take one extra cycle for each feature selected.

Bus Request and Grant

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

Operation

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

- Three-stating the data and address buses and the asynchronous memory control signals. The synchronous memory control signals can optionally be three-stated.
- Asserting the Bus Grant ($\overline{\text{BG}}$) signal.

The processor may halt program execution if the bus is granted to an external device and an instruction fetch or data read/write request is made to external memory. When the external device releases $\overline{\text{BR}}$, the processor deasserts BG and continues execution from the point at which it stopped.

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

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

19 CAN MODULE

This chapter describes the Controller Area Network (CAN) module. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples. Familiarity with the CAN standard is assumed. Refer to Version 2.0 of *CAN Specification* from Robert Bosch GmbH.

Overview

Key features of the CAN module are:

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

The CAN module is a low bit rate serial interface intended for use in applications where bit rates are typically up to 1Mbit/s. The CAN protocol incorporates a data CRC check, message error tracking and fault node confinement as means to improve network reliability to the level required for control applications.

Interface Overview

The interface to the CAN bus is a simple two-wire line. See Figure 19-1 for a symbolic representation of the CAN transceiver interconnection, and Figure 19-2 for a block diagram. The Blackfin processor's CANTX output and CANRX input pins are connected to an external CAN transceiver's TX and RX pins (respectively). The CANTX and CANRX pins operate with TTL levels and are appropriate for operation with CAN bus transceivers according to ISO/DIS 11898.

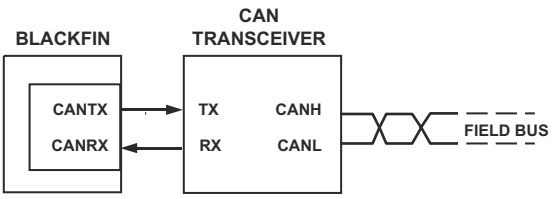


Figure 19-1. Representation of CAN Transceiver Interconnection

The CANRX and CANTX signals are multiplexed on Port C with GPIO pins. CAN functionality on the PC0 and PC1 pins is enabled by default, however, the pins can be freed as GPIO if software writes b#11 to the GPIO_C_CNFG register bits 1 and 0. CAN data is defined to be either *dominant* (logic 0) or *recessive* (logic 1). The default state of the CANTX output is recessive.

If PC1 is configured as GPIO, it functions as an input/open-drain output.

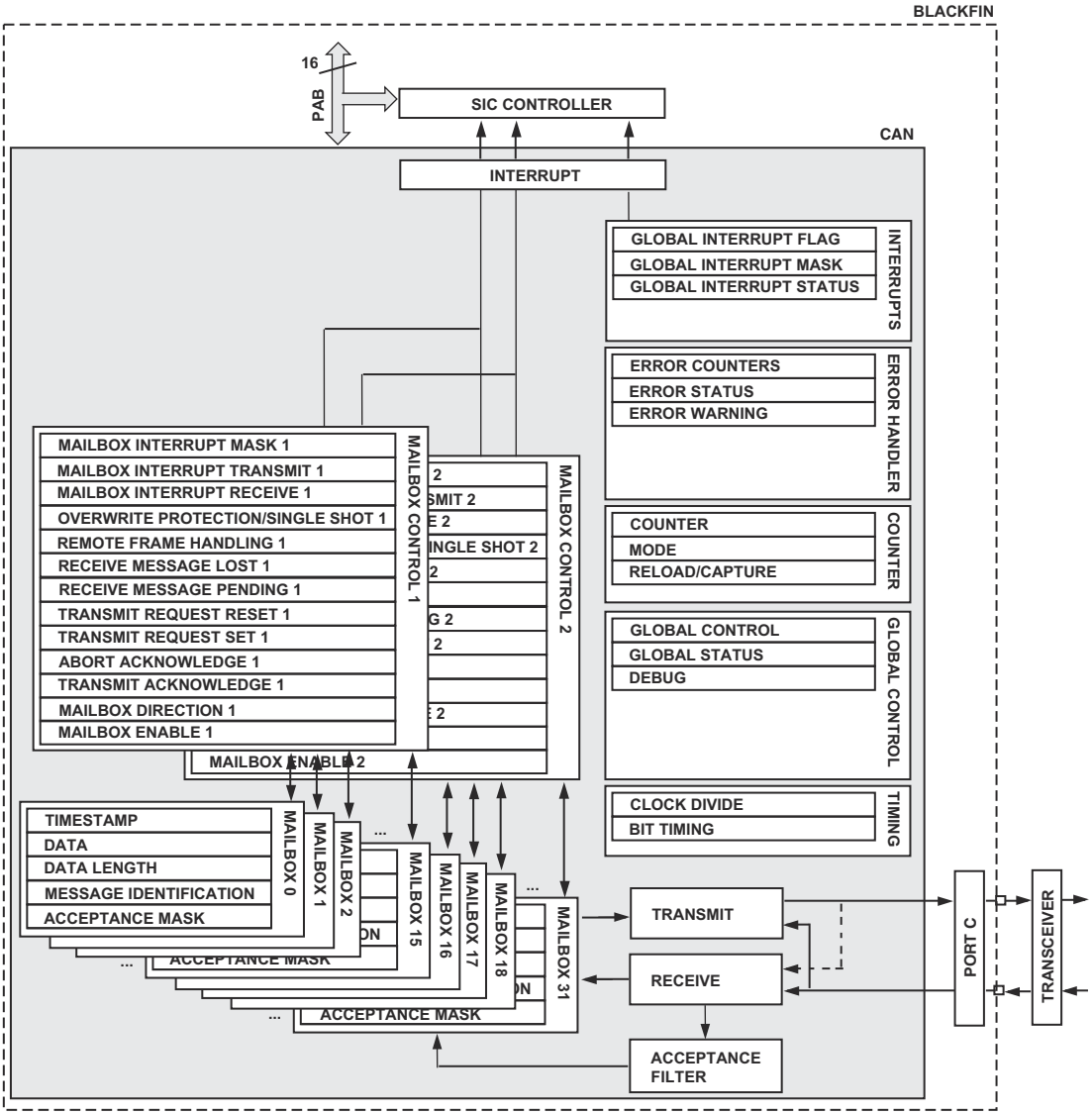


Figure 19-2. CAN Block Diagram

CAN Mailbox Area

The full-CAN controller features 32 message buffers, which are called mailboxes. Eight mailboxes are dedicated for message transmission, eight are for reception, and 16 are programmable in direction. Accordingly, the CAN module architecture is based around a 32-entry mailbox RAM. The mailbox is accessed sequentially by the CAN serial interface or the Blackfin core. Each mailbox consists of eight 16-bit control and data registers and two optional 16-bit acceptance mask registers, all of which must be configured before the mailbox itself is enabled. Since the mailbox area is implemented as RAM, the reset values of these registers are undefined. The data is divided into fields, which includes a message identifier, a time stamp, a byte count, up to 8 bytes of data, and several control bits. See Figure 19-3.

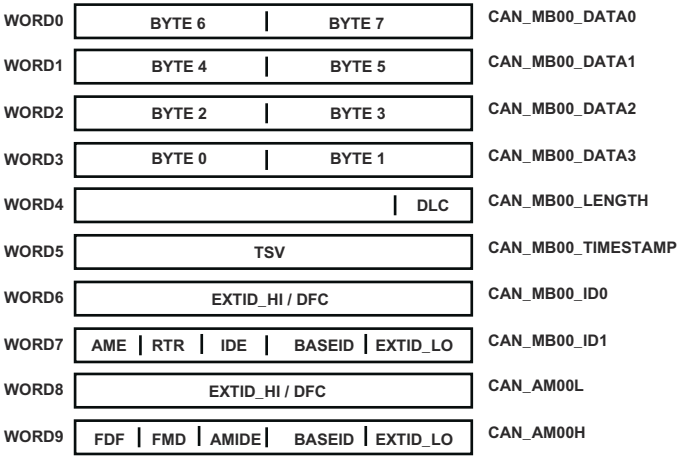


Figure 19-3. CAN Mailbox Area

The CAN mailbox identification (CAN_MBxx_ID0/1) register pair includes:

- The 29 bit identifier (base part BASEID plus extended part EXTID_LO/HI)
- The acceptance mask enable bit (AME)
- The remote transmission request bit (RTR)
- The identifier extension bit (IDE)

Do not write to the identifier of a message object while the mailbox is enabled for the CAN module (the corresponding bit in CAN_MCx is set).

The other mailbox area registers are:

- The data length code (DLC) in CAN_MBxx_LENGTH. The upper 12 bits of CAN_MBxx_LENGTH of each mailbox are marked as reserved. These 12 bits should always be set to 0.
- Up to eight bytes for the data field, sent MSB first from the CAN_MBxx_DATA3/2/1/0 registers, respectively, based on the number of bytes defined in the DLC. For example, if only one byte is transmitted or received (DLC = 1), then it is stored in the most significant byte of the CAN_MBxx_DATA3 register.
- Two bytes for the time stamp value (TSV) in the CAN_MBxx_TIMESTAMP register

The final registers in the mailbox area are the acceptance mask registers (CAN_AMxxH and CAN_AMxxL). The acceptance mask is enabled when the AME bit is set in the CAN_MBxx_ID1 register. If the “filtering on data field” option is enabled (DNM = 1 in the CAN_CONTROL register and FDF = 1 in the corresponding acceptance mask), the EXTID_HI[15:0] bits of CAN_MBxx_ID0 are reused as acceptance code (DFC) for the data field filtering. For more details, see “Receive Operation” on page 19-15 of this chapter.

CAN Mailbox Control

Mailbox control MMRs function as control and status registers for the 32 mailboxes. Each bit in these registers represents one specific mailbox. Since CAN MMRs are all 16 bits wide, pairs of registers are required to manage certain functionality for all 32 individual mailboxes. Mailboxes 0-15 are configured/monitored in registers with a suffix of 1. Similarly, mailboxes 16-31 use the same named register with a suffix of 2. For example, the CAN mailbox direction registers (`CAN_MDx`) would control mailboxes as shown in Figure 19-4.

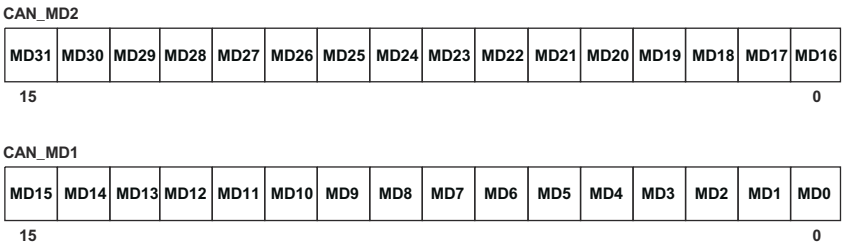


Figure 19-4. CAN Register Pairs

The mailbox control register area consists of these register pairs:

- `CAN_MC1` and `CAN_MC2` (mailbox enable registers)
- `CAN_MD1` and `CAN_MD2` (mailbox direction registers)
- `CAN_TA1` and `CAN_TA2` (transmit acknowledge registers)
- `CAN_AA1` and `CAN_AA2` (abort acknowledge registers)
- `CAN_TRS1` and `CAN_TRS2` (transmit request set registers)
- `CAN_TRR1` and `CAN_TRR2` (transmit request reset registers)
- `CAN_RMP1` and `CAN_RMP2` (receive message pending registers)
- `CAN_RML1` and `CAN_RML2` (receive message lost registers)

- CAN_RFH1 and CAN_RFH2 (remote frame handling registers)
- CAN_OPSS1 and CAN_OPSS2 (overwrite protection/single shot transmission registers)
- CAN_MBIM1 and CAN_MBIM2 (mailbox interrupt mask registers)
- CAN_MBTIF1 and CAN_MBTIF2 (mailbox transmit interrupt flag registers)
- CAN_MBRIF1 and CAN_MBRIF2 (mailbox receive interrupt flag registers)

Since mailboxes 24–31 support transmit operation only and mailboxes 0–7 are receive-only mailboxes, the lower eight bits in the "1" registers and the upper eight bits in the "2" registers are sometimes reserved or are restricted in their usage.

CAN Protocol Basics

Although the CANRX and CANTX pins are TTL-compliant signals, the CAN signals beyond the transceiver (see Figure 19-1 on page 19-2) have asymmetric drivers. A low state on the CANTX pin activates strong drivers while a high state is driven weakly. Consequently, active low is called the "dominant" state and active high is called "recessive." If the CAN module is passive, the CANTX pin is always high. If two CAN nodes transmit at the same time, dominant bits overwrite recessive bits.

The CAN protocol defines that all nodes trying to send a message on the CAN bus attempt to send a frame once the CAN bus becomes available. The start of frame indicator (SOF) signals the beginning of a new frame. Each CAN node then begins transmitting its message starting with the message ID. While transmitting, the CAN controller samples the CANRX pin to verify that the logic level being driven is the value it just placed on the CANTX pin. This is where the names for the logic levels apply. If a transmitting node places a recessive '1' on CANTX and detects a dominant '0' on

the CANRX pin, it knows that another node has placed a dominant bit on the bus, which means another node has higher priority. So, if the value sensed on CANRX is the value driven on CANTX, transmission continues, otherwise the CAN controller senses that it has lost arbitration and configuration determines what the next course of action is once arbitration is lost. See Figure 19-5 for more details regarding CAN frame structure.

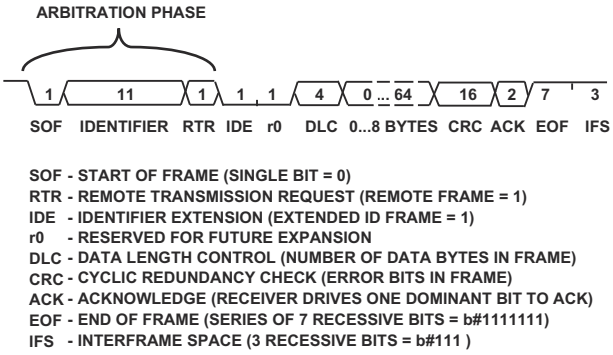


Figure 19-5. Standard CAN Frame

Figure 19-5 is a basic 11-bit identifier frame. After the SOF and identifier is the RTR bit, which indicates whether the frame contains data (data frame) or is a request for data associated with the message identifier in the frame being sent (remote frame).

Due to the inherent nature of the CAN protocol, a dominant bit in the RTR field wins arbitration against a remote frame request (RTR=1) for the same message ID, thereby defining a remote request to be lower priority than a data frame.

The next field of interest is the IDE. When set, it indicates that the message is an extended frame with a 29-bit identifier instead of an 11-bit identifier. In an extended frame, the first part of the message resembles Figure 19-6.

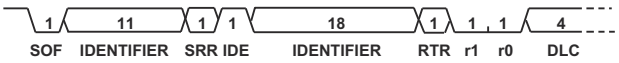


Figure 19-6. Extended CAN Frame

As could be concluded with regards to the RTR field, a dominant bit in the IDE field wins arbitration against an extended frame with the same lower 11-bits, therefore, standard frames are higher priority than extended frames. The substitute remote request bit (SRR, always sent as recessive), the reserved bits r0 and r1 (always sent as dominant), and the checksum (CRC) are generated automatically by the internal logic.

CAN Operation

The CAN controller is in configuration mode when coming out of processor reset or hibernate. It is only when the CAN is in configuration mode that hardware behavior can be altered. Before initializing the mailboxes themselves, the CAN bit timing must be set up to work on the CAN bus that the controller is expected to connect to.

Bit Timing

The CAN controller does not have a dedicated clock. Instead, the CAN clock is derived from the system clock (SCLK) based on a configurable number of time quanta. The Time Quantum (TQ) is derived from the formula TQ = (BRP+1)/SCLK, where BRP is the 10-bit BRP field in the CAN_CLOCK register. Although the BRP field can be set to any value, it is recommended that the value be greater than or equal to 4, as restrictions apply to the bit timing configuration when BRP is less than 4.

The CAN_CLOCK register defines the TQ value, and multiple time quanta make up the duration of a CAN bit on the bus. The CAN_TIMING register controls the nominal bit time and the sample point of the individual bits in the CAN protocol. Figure 19-7 shows the three phases of a CAN bit—the synchronization segment, the segment before the sample point, and the segment after the sample point.

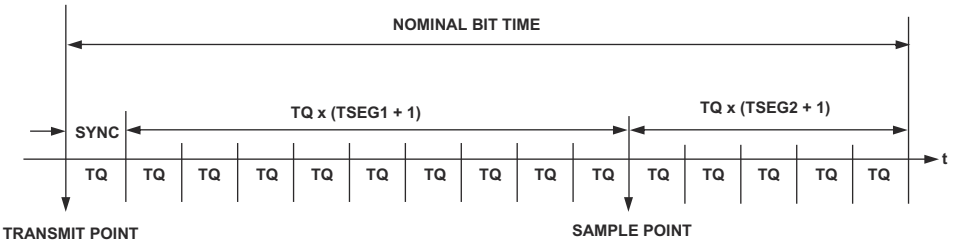


Figure 19-7. Three Phases of a CAN Bit

The synchronization segment is fixed to one TQ. It is required to synchronize the nodes on the bus. All signal edges are expected to occur within this segment.

The TSEG1 and TSEG2 fields of CAN_TIMING control how many TQs the CAN bits consist of, resulting in the CAN bit rate. The nominal bit time is given by the formula t_{BIT} = TQ x (1 + (1 + TSEG1) + (1 + TSEG2)). For safe receive operation on given physical networks, the sample point is programmable by the TSEG1 field. The TSEG2 field holds the number of TQs needed to complete the bit time. Often, best sample reliability is achieved with sample points in the high 80% range of the bit time. Never use sample points lower than 50%. Thus, TSEG1 should always be greater than or equal to TSEG2. To compute the sampling point, use the following equation:

(TSEG1 + 2) / (TSEG1 + TSEG2 + 3)

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 resynchronization attempt. The SJW value should not exceed TSEG2 or TSEG1. Therefore, the fundamental rule for writing CAN_TIMING is:

```
SJW <= TSEG2 <= TSEG1
```

In addition to this fundamental rule, phase segment 2 must also be greater than or equal to the Information Processing Time (IPT). This is the time required by the logic to sample CANRX input. On the Blackfin CAN module, this is 3 SCLK cycles. Because of this, restrictions apply to the minimal value of TSEG2 if the clock prescaler BRP is lower than 2. If BRP is set to 0, the TSEG2 field must be greater than or equal to 2. If the prescaler is set to 1, the minimum TSEG2 is 1.

All nodes on a CAN bus should use the same nominal bit rate.

With all the timing parameters set, the final consideration is how sampling is performed. The default behavior of the CAN controller is to sample the CAN bit once at the sampling point described by the CAN_TIMING register, controlled by the SAM bit. If the SAM bit is set, however, the input signal is oversampled three times at the SCLK rate. The resulting value is generated by a majority decision of the three sample values. Always keep the SAM bit cleared if the BRP value is less than 4.

Do not modify the CAN_CLOCK or CAN_TIMING registers during normal operation. Always enter configuration mode first. Writes to these registers have no effect if not in configuration or debug mode. If not coming out of

processor reset or hibernate, enter configuration mode by setting the CCR bit in the master control (CAN_CONTROL) register and poll the global CAN status (CAN_STATUS) register until the CCA bit is set.

If the TSEG1 field of the CAN_TIMING register is programmed to '0,' the module doesn't leave the configuration mode.

During configuration mode, the module is not active on the CAN bus line. The CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. After leaving the configuration mode, all CAN core internal registers and the CAN error counters are set to their initial values.

A software reset does not change the values of CAN_CLOCK and CAN_TIMING. Thus, an ongoing transfer via the CAN bus cannot be corrupted by changing the bit timing parameter or initiating the software reset (SRS = 1 in CAN_CONTROL).

Transmit Operation

Figure 19-8 shows the CAN transmit operation. Mailboxes 24-31 are dedicated transmitters. Mailboxes 8-23 can be configured as transmitters by writing 0 to the corresponding bit in the CAN_MDx register. After writing the data and the identifier into the mailbox area, the message is sent after mailbox n is enabled (MCn = 1 in CAN_MCx) and, subsequently, the corresponding transmit request bit is set (TRSn = 1 in CAN_TRSx).

When a transmission completes, the corresponding bits in the transmit request set register and in the transmit request reset register (TRRn in CAN_TRRx) are cleared. If transmission was successful, the corresponding bit in the transmit acknowledge register (TAn in CAN_TAx) is set. If the transmission was aborted due to lost arbitration or a CAN error, the corresponding bit in the abort acknowledge register (AAn in CAN_AAx) is set. A requested transmission can also be manually aborted by setting the corresponding TRRn bit in CAN_TRRx.

Multiple CAN_TRSx bits can be set simultaneously by software, and these bits are reset after either a successful or an aborted transmission. The TRSn bits can also be set by the CAN hardware when using the auto-transmit mode of the universal counter, when a message loses arbitration and the single-shot bit is not set (OPSSn = 0 in CAN_OPSSx), or in the event of a remote frame request. The latter is only possible for receive/transmit mailboxes if the automatic remote frame handling feature is enabled (RFHn = 1 in CAN_RFHx).

Special care should be given to mailbox area management when a TRSn bit is set. Write access to the mailbox is permissible with TRSn set, but changing data in such a mailbox may lead to unexpected data during transmission.

Enabling and disabling mailboxes has an impact on transmit requests. Setting the TRSn bit associated with a disabled mailbox may result in erroneous behavior. Similarly, disabling a mailbox before the associated TRSn bit is reset by the internal logic can cause unpredictable results.

Retransmission

Normally, the current message object is sent again after arbitration is lost or an error frame is detected on the CAN bus line. If there is more than one transmit message object pending, the message object with the highest mailbox is sent first (see Figure 19-8). The currently aborted transmission is restarted after any messages with higher priority are sent.

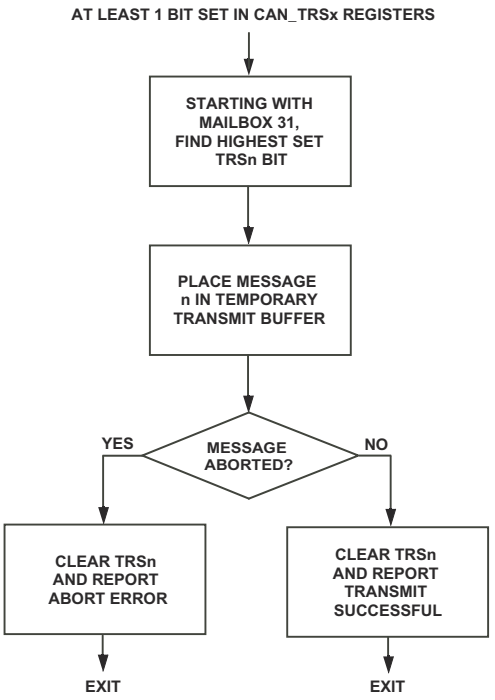


Figure 19-8. CAN Transmit Operation Flow Chart

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message in the buffer is not replaced until it is sent successfully, the arbitration on the CAN bus line is lost, or there is an error frame on the CAN bus line.

Single Shot Transmission

If the single shot transmission feature is used (`OPSSn = 1` in `CAN_OPSSx`), the corresponding `TRSn` bit is cleared after the message is successfully sent or if the transmission is aborted due to a lost arbitration or an error frame on

the CAN bus line. Thus, there is no further attempt to transmit the message again if the initial try failed, and the abort error is reported (AAn = 1 in CAN_AAx)

Auto-Transmission

In auto-transmit mode, the message in mailbox 11 can be sent periodically using the universal counter. This mode is often used to broadcast heartbeats to all CAN nodes. Accordingly, messages sent this way usually have high priority.

The period value is written to the CAN_UCRC register. When enabled in this mode (set UCCNF[3:0] = 0x3 in CAN_UCCNF), the counter (CAN_UCCNT) is loaded with the value in the CAN_UCRC register. The counter decrements at the CAN bit clock rate down to 0 and is then reloaded from CAN_UCRC. Each time the counter reaches a value of 0, the TRS11 bit is automatically set by internal logic, and the corresponding message from mailbox 11 is sent.

For proper auto-transmit operation, mailbox 11 must be configured as a transmit mailbox and must contain valid data (identifier, control bits, and data) before the counter first expires after this mode is enabled.

Receive Operation

The CAN hardware autonomously receives messages and discards invalid messages. Once a valid message has been successfully received, the receive logic interrogates all enabled receive mailboxes sequentially, from mailbox 23 down to mailbox 0, whether the message is of interest to the local node or not.

Each incoming data frame is compared to all identifiers stored in active receive mailboxes (MDn = 1 and MCn = 1) and to all active transmit mailboxes with the remote frame handling feature enabled (RFHn = 1 in CAN_RFHx).

The message identifier of the received message, along with the identifier extension (IDE) and remote transmission request (RTR) bits, are compared against each mailbox's register settings. 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. A one at the respective bit position in the CAN_AMxx mask registers means that the bit does not need to match when AME=1. In this way, a mailbox can accept a group of messages. Figure 19-9 shows the acceptance mask filtering logic.

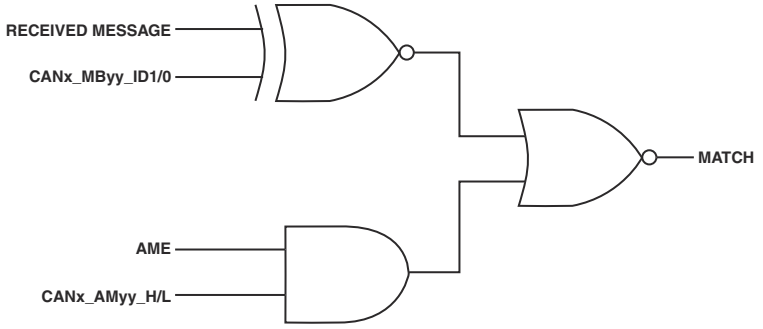


Figure 19-9. Bitwise Acceptance Mask Logic

Table 19-1. 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. Specifically, the DLC is extracted from the message and stored to the `CAN_MBxx_LENGTH` register, and the data bytes are placed into the appropriate `CAN_MBxx_DATAy` register(s). 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 19-10 illustrates the decision tree of the receive logic when processing the individual mailboxes.

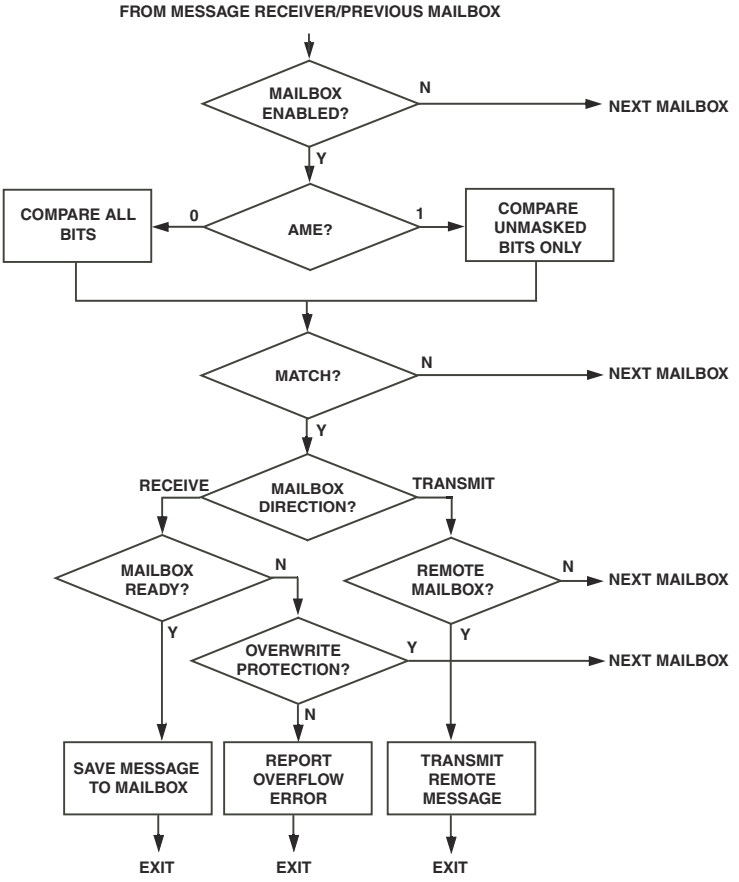


Figure 19-10. CAN Receive Operation Flow Chart

If a message is received for a mailbox and that mailbox still contains unread data (`RMPn = 1`), the user has to decide whether the old message should be overwritten or not. If `OPSSn = 0`, the receive message lost bit (`RMLn` in `CAN_RMLx`) is set and the stored message is overwritten. This results in the receive message lost interrupt being raised in the global CAN

interrupt status register (RMLIS = 1 in CAN_GIS). If OPSSn = 1, the next mailboxes are checked for another matching identifier. If no match is found, the message is discarded and the next message is checked.

If a receive mailbox is disabled, an ongoing receive message for that mailbox is lost even if a second mailbox is configured to receive the same identifier.

Data Acceptance Filter

If DeviceNet mode is enabled (DNM = 1 in CAN_CONTROL) and the mailbox is set up for filtering on data field, the filtering is done on the standard ID of the message and data fields. The data field filtering can be programmed for either the first byte only or the first two bytes, as shown in Table 19-2.

Table 19-2. Data Field Filtering

FDF Filter On Data Field	FMD Full Mask Data Field	Description
0	0	Do not allow filtering on the data field
0	1	Not allowed. FMD must be 0 if FDF is 0.
1	0	Filter on first data byte only
1	1	Filter on first two data bytes

If the FDF bit is set in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the data field mask (DFM[15:0]). If the FDF bit is cleared in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the extended identifier mask (EXTID_HI[15:0]).

Remote Frame Handling

Automatic handling of remote frames can be enabled for a transmit mailbox by setting the corresponding bit in the remote frame handling registers (CAN_RFHx).

Remote frames are data frames with no data field and the RTR bit set. The data length code of the data frame is equal to the DLC of the corresponding remote frame. A data length code can be programmed with values in the range of 0 to 15, but data length code values greater than 8 are considered as 8. A remote frame contains:

- the identifier bits
- the control field DLC
- the remote transmission request (RTR) bit

Only configurable mailboxes 8–23 can process remote frames, but all mail boxes can receive and transmit remote frame requests. When setup for automatic remote frame handling, the CAN_OPSSx register has no effect. All content of a mailbox is always overwritten by an incoming message.

If a remote frame is received, the DLC of the corresponding mailbox is overwritten with the received value.

Erroneous behavior may result when the remote frame handling bit (RFHn) is changed and the corresponding mailbox is currently processed. To avoid the risk of inconsistent messages, it is recommended to temporarily disable the mailbox while its data registers are updated. See section "Temporarily Disabling Mailboxes" on page 19-23.

Watchdog Mode

Watchdog mode is used to make sure messages are received periodically. It is often used to observe whether or not a certain node on the network is alive and functioning properly, and, if not, to detect and manage its failure case accordingly.

Upon programming the universal counter to watchdog mode (set UCCNF[3:0] = 0x2 in CAN_UCCNF), the counter in the CAN_UCCNT register is loaded with the predefined value contained in the CAN universal counter reload/capture register (CAN_UCRC). This counter then decrements at the CAN bit rate. If the UCCT and UCRC bits in the CAN_UCCNF register are set and a message is received in mailbox 4 before the counter counts down to 0, the counter is reloaded with the CAN_UCRC contents. If the counter has counted down to 0 without receiving a message in mailbox 4, the UCEIS bit in the global CAN interrupt status (CAN_GIS) register is set, and the counter is automatically reloaded with the contents of the CAN_UCRC register. If an interrupt is desired, the UCEIM bit in the CAN_GIM register must also be set. With the mask bit set, when a watchdog interrupt occurs, the UCEIF bit in the CAN_GIF register is also set.

The counter can be reloaded with the contents of CAN_UCRC or disabled by writing to the CAN_UCCNF register.

The time period it takes for the watchdog interrupt to occur is controlled by the value written into the CAN_UCRC register by the user.

Time Stamps

To get an indication of the time of reception or the time of transmission for each message, program the CAN universal counter to time stamp mode (set UCCNF[3:0] = 0x1 in CAN_UCCNF). The value of the 16-bit free-running counter (CAN_UCCNT) is then written into the CAN_MBxx_TIMESTAMP register of the corresponding mailbox when a received message has been stored or a message has been transmitted.

The time stamp value is captured at the sample point of the start of frame (SOF) bit of each incoming or outgoing message. Afterwards, this time stamp value is copied to the CAN_MBxx_TIMESTAMP register of the corresponding mailbox.

If the mailbox is configured for automatic remote frame handling, the time stamp value is written for transmission of a data frame (mailbox configured as transmit) or the reception of the requested data frame (mailbox configured as receive).

The counter can be cleared (set UCRC bit to 1) or disabled (set UCE bit to 0) by writing to the CAN_UCCNF register. The counter can also be loaded with a value by writing to the counter register itself (CAN_UCCNT).

It is also possible to clear the counter (CAN_UCCNT) by reception of a message in mailbox number 4 (synchronization of all time stamp counters in the system). This is accomplished by setting the UCCT bit in the CAN_UCCNF register.

An overflow of the counter sets a bit in the global CAN interrupt status register (UCEIS in the CAN_GIS register). A global CAN interrupt can optionally occur by unmasking the bit in the global CAN interrupt mask register (UCEIM in the CAN_GIM register). If the interrupt source is unmasked, a bit in the global CAN interrupt flag register is also set (UCEIF in the CAN_GIF register).

Temporarily Disabling Mailboxes

If a mailbox is enabled and configured as "transmit," write accesses to the data field should be guarded to avoid transmission of inconsistent messages. Special care must be taken if the mailbox is transmitting (or attempting to transmit) repeatedly. Also, if this mailbox is used for automatic remote frame handling, the data field must be updated without losing an incoming remote request frame and without sending inconsistent data. Therefore, the CAN controller allows for temporary mailbox disabling, which can be enabled by programming the mailbox temporary disable register (CAN_MBTD).

The pointer to the requested mailbox must be written to the TDPTR[4:0] bits of the CAN_MBTD register and the mailbox temporary disable request bit (TDR) must be set. The corresponding mailbox temporary disable flag (TDA) is subsequently set by the internal logic.

If a mailbox is configured as "transmit" (MDn = 0) and TDA is set, the content of the data field of that mailbox can be updated. If there is an incoming remote request frame while the mailbox is temporarily disabled, the corresponding transmit request set bit (TRSn) is set by the internal logic and the data length code of the incoming message is written to the corresponding mailbox. However, the message being requested is not sent until the temporary disable request is cleared (TDR = 0). Similarly, all transmit requests for temporarily disabled mailboxes are ignored until TDR is cleared. Additionally, transmission of a message is immediately aborted if the mailbox is temporarily disabled and the corresponding TRRn bit for this mailbox is set.

If a mailbox is configured as "receive" (MDn = 1), the temporary disable flag is set and the mailbox is not processed. If there is an incoming message for the mailbox n being temporarily disabled, the internal logic waits until the reception is complete or there is an error on the CAN bus to set TDA. Once TDA is set, the mailbox can then be completely disabled (MCn = 0) without the risk of losing an incoming frame. The temporary disable request (TDR) bit must then be reset as soon as possible.

When TDA is set for a given mailbox, only the data field of that mailbox can be updated. Accesses to the control bits and the identifier are denied.

Functional Operation

The following sections describe the functional operation of the CAN module, including interrupts, the event counter, warnings and errors, debug features, and low power features.

CAN Interrupts

The CAN module provides three independent interrupts: two mailbox interrupts (mailbox receive interrupt MBRIRQ and mailbox transmit interrupt MBTIRQ) and the global 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 CAN_MBIMx.

If a mailbox is configured as a receive mailbox, the corresponding receive interrupt flag is set (MBRIFn = 1 in CAN_MBRIFx) after a received message is stored in mailbox n (RMPn = 1 in CAN_RMPx). If the automatic remote frame handling feature is used, the receive interrupt flag is set after the requested data frame is stored in the mailbox. If any MBRIFn bits are set in CAN_MBRIFx, the MBRIRQ interrupt output is raised in CAN_INTR. In order to clear the MBRIRQ interrupt request, all of the set MBRIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBRIFx. 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 CAN_MBTIFx) after the message in mailbox n is sent correctly (TAn = 1 in CAN_TAx). The TAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0). If the automatic remote frame handling feature is used, the transmit interrupt flag is set after the requested data frame is sent from the mailbox. If any MBTIFn bits are set in CAN_MBTIFx, the MBTIRQ interrupt output is raised in CAN_INTR. In order to clear the MBTIRQ interrupt request, all of the set MBTIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBTIFx. Additionally, software must clear the associated TAn bit or set the associated TRSn bit to clear the interrupt source that asserts the MBTIFn bit.

Global CAN Interrupt

The global CAN interrupt logic is implemented with three registers—the global CAN interrupt mask register (CAN_GIM), where each interrupt source can be enabled or disabled separately; the global CAN interrupt status register (CAN_GIS); and the global CAN interrupt flag register (CAN_GIF). The interrupt mask bits only affect the content of the global CAN interrupt flag register (CAN_GIF). If the mask bit is not set, the corresponding flag bit is not set when the event occurs. The interrupt status bits in the global 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 CAN_GIF register is set. The GIRQ bit remains set as long as at least one bit in the interrupt flag register CAN_GIF is set. All bits in the interrupt status and in the interrupt flag registers remain set until cleared by software or a software reset has occurred.

In the ISR, the interrupt latch should be cleared by a W1C operation to the corresponding bit of the CAN_GIS register. This clears the related bits of both the CAN_GIS and CAN_GIF registers.

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 has been received for a mailbox that currently contains unread data. At least one bit in the receive message lost register (CAN_RMLx) is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_RMLx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_RMLx is set.

- **Abort acknowledge interrupt** (AAIM, AAIS, AAIF)

  At least one AAn bit in the abort acknowledge registers CAN_AAx is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_AAx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_AAx is set. The AAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0).

- **Access to unimplemented address interrupt** (UIAIM, UIAIS, UIAIF)

  There was a CPU access to an address which is not implemented in the controller module.

- **Wakeup interrupt** (WUIM, WUIS, WUIF)

  The CAN module has left the sleep mode because of detected activity on the CAN bus line.

- **Bus-Off interrupt** (BOIM, BOIS, BOIF)

  The CAN module has entered the bus-off state. This interrupt source is active if the status of the CAN core changes from normal operation mode to the bus-off mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the bus-off mode is still active, this bit is not set again. If the module leaves the bus-off mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error-Passive interrupt** (EPIM, EPIS, EPIF)

  The CAN module has entered the error-passive state. This interrupt source is active if the status of the CAN module changes from the error-active mode to the error-passive mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the error-passive mode is still active, this bit is not set again. If the module leaves the error-passive mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning receive interrupt** (EWRIM, EWRIS, EWRIF)

  The CAN receive error counter (RXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning transmit interrupt** (EWTIM, EWTIS, EWTIF)

  The CAN transmit error counter (TXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

Event Counter

For diagnostic functions, it is possible to use the universal counter as an event counter. The counter can be programmed in the 4-bit UCCNF[3:0] field of CAN_UCCNF to increment on one of these conditions:

- UCCNF[3:0] = 0x6 – CAN error frame. Counter is incremented if there is an error frame on the CAN bus line.
- UCCNF[3:0] = 0x7 – CAN overload frame. Counter is incremented if there is an overload frame on the CAN bus line.
- UCCNF[3:0] = 0x8 – Lost arbitration. Counter is incremented every time arbitration on the CAN line is lost during transmission.
- UCCNF[3:0] = 0x9 – Transmission aborted. Counter is incremented every time arbitration is lost or a transmit request is cancelled (AAn is set).
- UCCNF[3:0] = 0xA – Transmission succeeded. Counter is incremented every time a message sends without detected errors (TAn is set).
- UCCNF[3:0] = 0xB – Receive message rejected. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because there is no matching identifier found.
- UCCNF[3:0] = 0xC – Receive message lost. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because the mailbox contains unread data (RMLn is set).
- UCCNF[3:0] = 0xD – Message received. Counter is incremented every time a message is received without detected errors, whether the received message is rejected or stored in a mailbox.

- UCCNF[3:0] = 0xE – Message stored. Counter is incremented every time a message is received without detected errors, has an identifier that matches an enabled receive mailbox, and is stored in the receive mailbox (RMPn is set).
- UCCNF[3:0] = 0xF – Valid message. Counter is incremented every time a valid transmit or receive message is detected on the CAN bus line.

CAN Warnings and Errors

CAN warnings and errors are controlled using the CAN_CEC register, the CAN_ESR register, and the CAN_EWR register.

Programmable Warning Limits

It is possible to program the warning level for EWTIS (error warning transmit interrupt status) and EWRIS (error warning receive interrupt status) separately by writing to the error warning level error count fields for receive (EWLREC) and transmit (EWLTEC) in the CAN error counter warning level (CAN_EWR) register. After powerup reset, the CAN_EWR register is set to the default warning level of 96 for both error counters. After software reset, the content of this register remains unchanged.

CAN Error Handling

Error management is an integral part of the CAN standard. Five different kinds of bus errors may occur during transmissions:

- **Bit error**

  A bit error can be detected by the transmitting node only. Whenever a node is transmitting, it continuously monitors its receive pin (CANRX) and compares the received data with the transmitted data. During the arbitration phase, the node simply postpones the transmission if the received and transmitted data do not match. However, after the arbitration phase (that is, once the RTR bit has been sent successfully), a bit error is signaled any time the value on CANRX does not equal what is being transmitted on CANTX.

- **Form error**

  A form error occurs any time a fixed-form bit position in the CAN frame contains one or more illegal bits, that is, when a dominant bit is detected at a delimiter or end-of-frame bit position.

- **Acknowledge error**

  An acknowledge error occurs whenever a message has been sent and no receivers drive an acknowledge bit.

- **CRC error**

  A CRC error occurs whenever a receiver calculates the CRC on the data it received and finds it different than the CRC that was transmitted on the bus itself.

- **Stuff error**

  The CAN specification requires the transmitter to insert an extra stuff bit of opposite value after 5 bits have been transmitted with the same value. The receiver disregards the value of these stuff bits.

However, it takes advantage of the signal edge to resynchronize itself. A stuff error occurs on receiving nodes whenever the 6th consecutive bit value is the same as the previous five bits.

Once the CAN module detects any of the above errors, it updates the error status register CAN_ESR as well as the error counter register CAN_CEC. In addition to the standard errors, the CAN_ESR register features a flag that signals when the CANRX pin sticks at dominant level, indicating that shorted wires are likely.

Error Frames

It is of central importance that all nodes on the CAN bus ignore data frames that one single node failed to receive. To accomplish this, every node sends an error frame as soon as it has detected an error. See Figure 19-11.

Once a device has detected an error, it still completes the ongoing bit and initiates an error frame by sending six dominant and eight recessive bits to the bus. This is a violation to the bit stuffing rule and informs all nodes that the ongoing frame needs to be discarded.

All receivers that did not detect the transmission error in the first instance now detect a stuff bit error. The transmitter may detect a normal bit error sooner. It aborts the transmission of the ongoing frame and tries sending it again later.

Finally, all nodes on the bus have detected an error. Consequently, all of them send 6 dominant and 8 recessive bits to the bus as well. The resulting error frame consists of two different fields. The first field is given by the superposition of error flags contributed from the different stations, which is a sequence of 6 to 12 dominant bits. The second field is the error delimiter and consists of 8 recessive bits indicating the end of frame.

For CRC errors, the error frame is initiated at the end of the frame, rather than immediately after the failing bit.

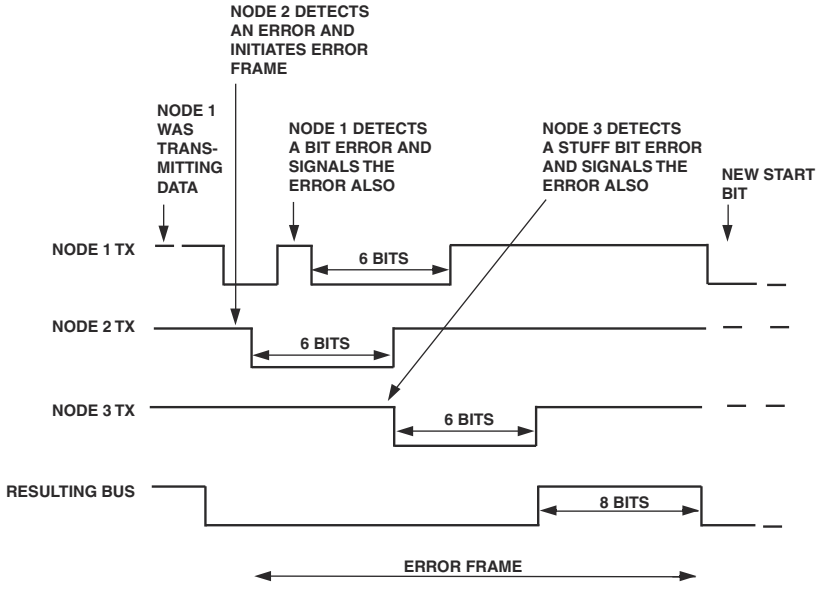


Figure 19-11. CAN Error Scenario Example

After having received 8 recessive bits, every node knows that the error condition has been resolved and starts transmission if messages are pending. The former transmitter that had to abort its operation must win the new arbitration again, otherwise its message is delayed as determined by priority.

Because the transmission of an error frame destroys the frame under transmission, a faulty node erroneously detecting an error can block the bus. Because of this, there are two node states which determine a node's right to signal an error—error active and error passive. Error active nodes are those which have an error detection rate below a certain limit. These nodes drive an 'active error flag' of 6 dominant bits.

Nodes with a higher error detection rate are suspected of having a local problem and, therefore, have a limited right to signal errors. These error passive nodes drive a 'passive error flag' consisting of 6 recessive bits.

Thus, an error passive transmitting node is still able to inform the other nodes about the abortion of a self-transmitted frame, but it is no longer able to destroy correctly received frames of other nodes.

Error Levels

The CAN specification requires each node in the system to operate in one of three levels. See Table 19-3. This prevents nodes with high error rates from blocking the entire network, as the errors might be caused by local hardware. The Blackfin CAN module provides an error counter for transmit (TEC) and an error counter for receive (REC). The CAN error count register CAN_CEC houses each of these 8-bit counters.

After initialization, both the TEC and the REC counters are 0. Each time a bus error occurs, one of the counters is incremented by either 1 or 8, depending on the error situation (documented in Version 2.0 of *CAN Specification*). Successful transmit and receive operations decrement the respective counter by 1.

If either of the error counters exceeds 127, the CAN module goes into a passive state and the CAN error passive mode (EP) bit in CAN_STATUS is set. Then, it is not allowed to send any more active error frames. However, it is still allowed to transmit messages and to signal passive error frames in case the transmission fails because of a bit error.

If one of the counters exceeds 255 (that is, when the 8-bit counters overflow), the CAN module is disconnected from the bus. It goes into bus off mode and the CAN error bus off mode (EBO) bit is set in CAN_STATUS. Software intervention is required to recover from this state unless the ABO bit in the CANx_CONTROL register has been enabled.

Table 19-3. CAN Error Level Description

Level	Condition	Description
Error active	Transmit and receive error counters < 128	This is the initial condition level. As long as errors stay below 128, the node will drive active error flags during error frames.
Error passive	Transmit or receive error counters ≥ 128, but < 256	Errors have accumulated to a level which requires the node to drive passive error flags during error frames.
Bus off	Transmit or receive error counters ≥ 256	CAN module goes into bus off mode

In addition to these levels, the CAN module also provides a warning mechanism, which is an enhancement to the CAN specification. There are separate warnings for transmit and receive. By default, when one of the error counters exceeds 96, a warning is signaled and is represented in the CAN_STATUS register by either the CAN receive warning flag (WR) or CAN transmit warning flag (WT) bits. The error warning level can be programmed using the error warning register, CAN_EWR. More information is available on page 19-87.

Additionally, interrupts can occur for all of these levels by unmasking them in the global CAN interrupt mask register (CAN_GIM) shown on page 19-49. The interrupts include the bus off interrupt (BOIM), the error-passive interrupt (EPIM), the error warning receive interrupt (EWRIM), and the error warning transmit interrupt (EWTIM).

During the bus off recovery sequence, the configuration mode request bit in the CAN_CONTROL register is set by the internal logic (CCR = 1), thus the CAN core module does not automatically come out of the bus off mode. The CCR bit cannot be reset until the bus off recovery sequence is finished.

This behavior can be over-ridden by setting the auto-bus on (ABO) bit in the CAN_CONTROL register. After exiting the bus off or configuration modes, the CAN error counters are reset.

Debug and Test Modes

The CAN module contains test mode features that aid in the debugging of the CAN software and system. Listing 19-1 provides an example of enabling CAN debug features.

When these features are used, the CAN module may not be compliant to the CAN specification. All test modes should be enabled or disabled only when the module is in configuration mode (CCA = 1 in the CAN_STATUS register) or in suspend mode (CSA = 1 in CAN_STATUS).

The CDE bit is used to gain access to all of the debug features. This bit must be set to enable the test mode, and must be written first before subsequent writes to the CAN_DEBUG register. When the CDE bit is cleared, all debug features are disabled.

Listing 19-1. Enabling CAN Debug Features in C

```
#include <cdefBF539.h>
/* Enable debug mode, CDE must be set before other flags can be
changed in register */
*pCAN_DEBUG |= CDE ;

/* Set debug flags */
*pCAN_DEBUG &= ~DTO ;
*pCAN_DEBUG |= MRB | MAA | DIL ;

/* Run test code */

/* Disable debug mode */
*pCAN_DEBUG &= ~CDE ;
```

When the CDE bit is set, it enables writes to the other bits of the CAN_DEBUG register. It also enables these features, which are not compliant with the CAN standard:

- Bit timing registers can be changed anytime, not only during configuration mode. This includes the CAN_CLOCK and CAN_TIMING registers.
- Allows write access to the read-only transmit/receive error counter register CAN_CEC.

The mode read back bit (MRB) is used to enable the read back mode. In this mode, a message transmitted on the CAN bus (or via an internal loop back mode) is received back directly to the internal receive buffer. After a correct transmission, the internal logic treats this as a normal receive message. This feature allows the user to test most of the CAN features without an external device.

The mode auto acknowledge bit (MAA) allows the CAN module to generate its own acknowledge during the ACK slot of the CAN frame. No external devices or connections are necessary to read back a transmit message. In this mode, the message that is sent is automatically stored in the internal receive buffer. In auto acknowledge mode, the module itself transmits the acknowledge. This acknowledge can be programmed to appear on the CANTX pin if DIL=1 and DTO=0. If the acknowledge is only going to be used internally, then these test mode bits should be set to DIL=0 and DTO=1.

The disable internal loop bit (DIL) is used to internally enable the transmit output to be routed back to the receive input.

The disable transmit output bit (DTO) is used to disable the CANTX output pin. When this bit is set, the CANTX pin continuously drives recessive bits.

The disable receive input bit (DRI) is used to disable the CANRX input. When set, the internal logic receives recessive bits or receives the internally generated transmit value in the case of the internal loop enabled (DIL=0). In either case, the value on the CANRX input pin is ignored.

The disable error counters bit (DEC) is used to disable the transmit and receive error counters in the CAN_CEC register. When this bit is set, the CAN_CEC holds its current contents and is not allowed to increment or decrement the error counters. This mode does not conform to the CAN specification.

Writes to the error counters should be in debug mode only. Write access during reception may lead to undefined values. The maximum value which can be written into the error counters is 255. Thus, the error counter value of 256 which forces the module into the bus off state can not be written into the error counters.

Table 19-4 shows several common combinations of test mode bits.

Table 19-4. CAN Test Modes

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
X	X	X	X	X	0	Normal mode, not debug mode.
0	X	X	X	X	X	No read back of transmit message.
1	0	1	0	0	1	Normal transmission on CAN bus line. Read back. External acknowledge from external device required.
1	1	1	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is enabled.

Table 19-4. CAN Test Modes (Cont'd)

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
1	1	0	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input and internal loop are enabled (internal OR of TX and RX).
1	1	0	0	1	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is ignored. Internal loop is enabled
1	1	0	1	1	1	No transmission on CAN bus line. Read back. No external acknowledge required. Neither transmit message nor acknowledge are transmitted on CANTX. CANRX input is ignored. Internal loop is enabled.

Low Power Features

The Blackfin processor provides a low power hibernate state, and the CAN module includes built-in sleep and suspend modes to save power. The behavior of the CAN module in these three modes is described in the following sections.

CAN Built-In Suspend Mode

The most modest of power savings modes is the suspend mode. This mode is entered by setting the suspend mode request (CSR) bit in the CAN_CONTROL register. The module enters the suspend mode after the current operation of the CAN bus is finished, at which point the internal logic sets the suspend mode acknowledge (CSA) bit in CAN_STATUS. Once this mode is entered, the module is no longer active on the CAN bus line, slightly reducing power consumption. When the CAN module is in suspend mode, the CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. The content of the CAN error counters remains unchanged.

The suspend mode can subsequently be exited by clearing the CSR bit in CAN_CONTROL. The only differences between suspend mode and configuration mode are that writes to the CAN_CLOCK and CAN_TIMING registers are still locked in suspend mode and the CAN control and status registers are not reset when exiting suspend mode.

CAN Built-In Sleep Mode

The next level of power savings can be realized by using the CAN module's built-in sleep mode. This mode is entered by setting the sleep mode request (SMR) bit in the CAN_CONTROL register. The module enters the sleep mode after the current operation of the CAN bus is finished. Once this mode is entered, many of the internal CAN module clocks are shut off, reducing power consumption, and the sleep mode acknowledge (SMACK) bit is set in CAN_INTR. When the CAN module is in sleep mode, all register reads return the contents of CAN_INTR instead of the usual contents. All register writes, except to CAN_INTR, are ignored in sleep mode.

A small part of the module is clocked continuously to allow for wakeup out of sleep mode. A write to the CAN_INTR register ends sleep mode. If the WBA bit in the CAN_CONTROL register is set before entering sleep mode, a dominant bit on the CANRX pin also ends sleep mode.

CAN Wakeup From Hibernate State

For greatest power savings, the Blackfin processor provides a hibernate state, where the internal voltage regulator shuts off the internal power supply to the chip, turning off the core and system clocks in the process. In this mode, the only power drawn (< 50μA) is that used by the regulator circuitry awaiting any of the possible hibernate wakeup events. One such event is a wakeup due to CAN bus activity. After hibernation, the CAN module must be re-initialized.

For low power designs, the external CAN bus transceiver is typically put into standby mode via one of the Blackfin processor's general purpose I/O pins. While in standby mode, the CAN transceiver continually drives the recessive logic '1' level onto the CANRX pin. If the transceiver then senses CAN bus activity, it will, in turn, drive the CANRX pin to the dominant logic '0' level. This signals to the Blackfin processor that CAN bus activity has been detected. If the internal voltage regulator is programmed to recognize CAN bus activity as an event to exit hibernate state, the part responds appropriately. Otherwise, the activity on the CANRX pin has no effect on the processor state.

To enable this functionality, the voltage regulator 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.

Register Definitions

The following sections describe the CAN controller register definitions.

Table 19-5 through Table 19-9 show the functions of the CAN controller registers.

Table 19-5. Global CAN Register Mapping

Register Name	Function	Notes
CAN_CONTROL	Master control register	Reserved bits 15:8 and 3 must always be written as '0'
CAN_STATUS	Global CAN status register	Write accesses have no effect
CAN_DEBUG	CAN debug register	Use of these modes is not CAN-compliant
CAN_CLOCK	CAN clock register	Accessible only in configuration mode
CAN_TIMING	CAN timing register	Accessible only in configuration mode
CAN_INTR	CAN interrupt register	Reserved bits 15:8 and 5:4 must always be written as '0'
CAN_GIM	Global CAN interrupt mask register	Bits 15:11 and 9 are reserved
CAN_GIS	Global CAN interrupt status register	Bits 15:11 and 9 are reserved
CAN_GIF	Global CAN interrupt flag register	Bits 15:11 and 9 are reserved

Table 19-6. CAN Mailbox/Mask Register Mapping

Register Name	Function	Notes
CAN_AMxxH/L	Acceptance mask registers	Do not write when mailbox MBxx is enabled
CAN_MBxx_ID1/0	Mailbox word 7/6 register	Do not write when mailbox MBxx is enabled
CAN_MBxx_TIMESTAMP	Mailbox word 5 register	Holds timestamp information when time-stamp mode is active
CAN_MBxx_LENGTH	Mailbox word 4 register	Values greater than 8 are not allowed. Bits 15:4 are reserved
CAN_MBxx_DATA3/2/1/0	Mailbox word 3/2/1/0 register	Software controls reading correct data based on DLC

Table 19-7. CAN Mailbox Control Register Mapping

Register Name	Function	Notes
CAN_MCx	Mailbox configuration registers	Always disable before modifying mailbox area or direction
CAN_MDx	Mailbox direction registers	Never change MDn direction when mailbox n is enabled. MD[31:24] and MD[7:0] are read only
CAN_RMPx	Receive message pending registers	Clearing RMPn bits also clears corresponding RMLn bits
CAN_RMLx	Receive message lost registers	Write accesses have no effect
CAN_OPSSx	Overwrite protection or single-shot transmission register	Function depends on mailbox direction. Has no effect when RFHn = 1. Do not modify OPSSn bit if mailbox n is enabled
CAN_TRSx	Transmission request set registers	May by set by internal logic under certain circumstances. TRS[7:0] are read-only
CAN_TRRx	Transmission request reset registers	TRRn bits must not be set if mailbox n is disabled or TRSn = 0

Table 19-7. CAN Mailbox Control Register Mapping (Cont'd)

Register Name	Function	Notes
CAN_AAx	Abort acknowledge registers	AAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_TAx	Transmission acknowledge registers	TAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_MBTD	Temporary mailbox disable feature register	Allows safe access to data field of an enabled mailbox
CAN_RFHx	Remote frame handling registers	Available only to configurable mailboxes 23:8. RFH[31:24] and RFH[7:0] are read-only
CAN_MBIMx	Mailbox interrupt mask registers	Mailbox interrupts are raised only if these bits are set
CAN_MBTIFx	Mailbox transmit interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBTIFn = 1 results in MBRIFn = 1 and MBTIFn = 0
CAN_MBRIFx	Mailbox receive interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBRIFn = 1 results in MBTIFn = 1 and MBRIFn = 0

Table 19-8. CAN Universal Counter Register Mapping

Register Name	Function	Notes
CAN_UCCNF	Universal counter mode register	Bits 15:8 and bit 4 are reserved
CAN_UCCNT	Universal counter register	Counts up or down based on universal counter mode
CAN_UCRC	Universal counter reload/capture register	In timestamp mode, holds time of last successful transmit or receive

Table 19-9. CAN Error Register Mapping

Register Name	Function	Notes
CAN_CEC	CAN error counter register	Undefined while in bus off mode, not affected by software reset
CAN_ESR	Error status register	Only the first error is stored. SA0 flag is cleared by recessive bit on CAN bus
CAN_EWR	CAN error counter warning level register	Default is 96 for each counter

Global CAN Registers

Figure 19-12 through Figure 19-20 on page 19-50 show the global CAN registers.

CAN_CONTROL Master Control Register

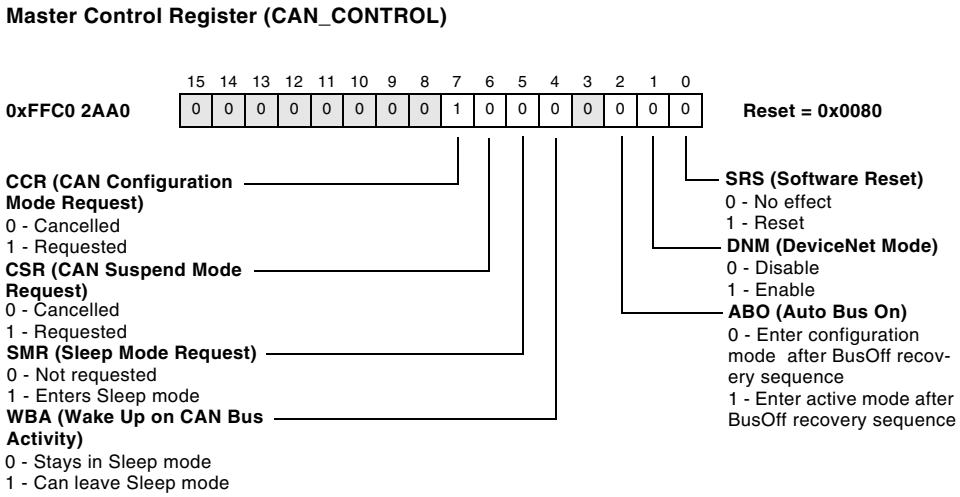


Figure 19-12. Master Control Register

CAN_STATUS Global CAN Status Register

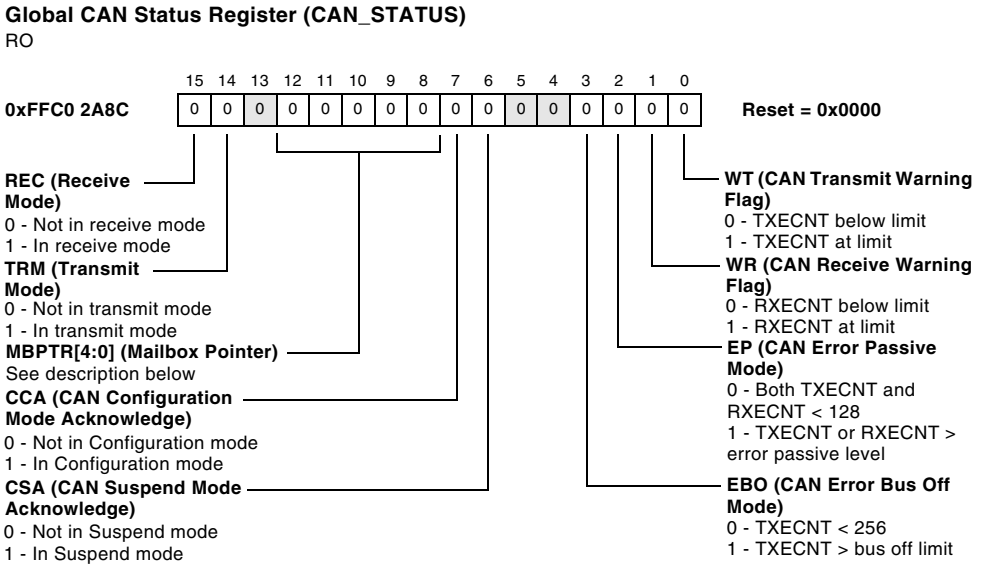


Figure 19-13. Global CAN Status Register

- **Mail box pointer** (`MBPTR[4:0]`)

  Represents the mailbox number of the current transmit message. After a successful transmission, these bits remain unchanged.

  b#11111 The message of mailbox 31 is currently being processed.

  …

  …

  …

  b#00000 The message of mailbox 0 is currently being processed.

CAN_DEBUG Register

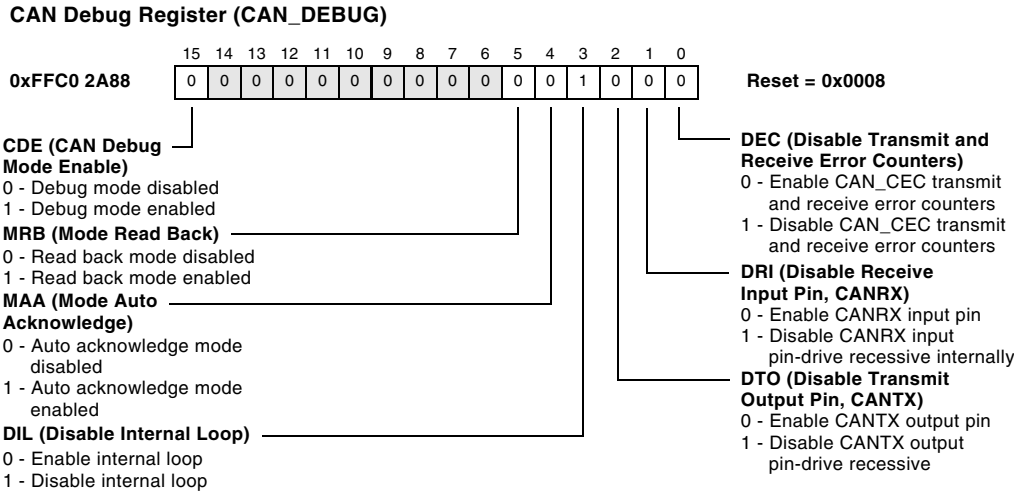


Figure 19-14. CAN Debug Register

CAN_CLOCK Register

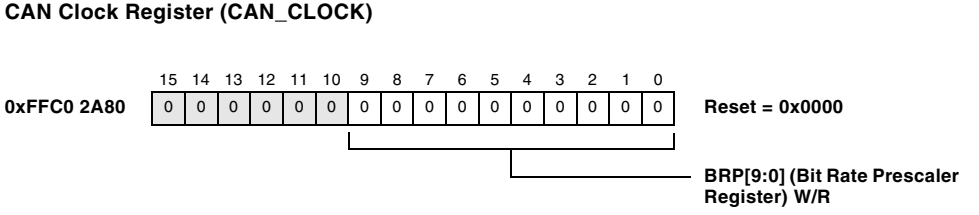


Figure 19-15. CAN Clock Register

CAN_TIMING Register

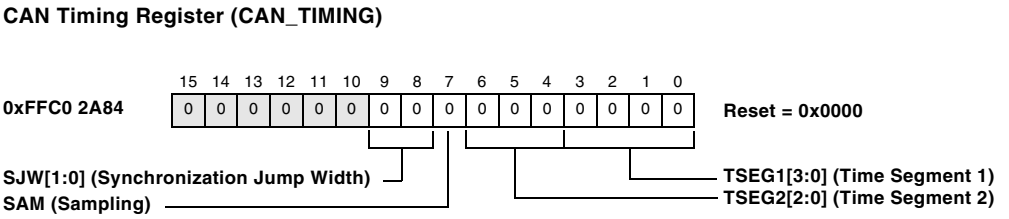


Figure 19-16. CAN Timing Register

CAN_INTR Interrupt Pending Register

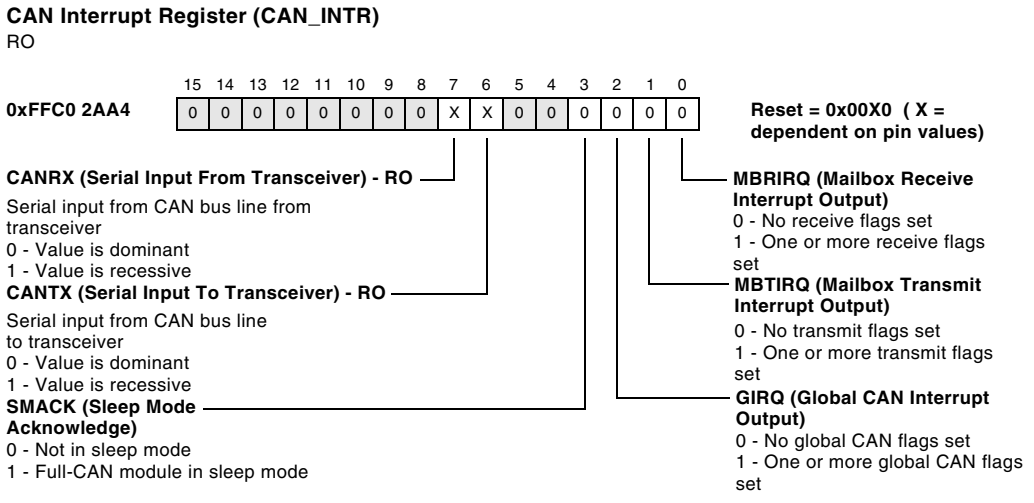


Figure 19-17. CAN Interrupt Register

CAN_GIM Global CAN Interrupt Mask Register

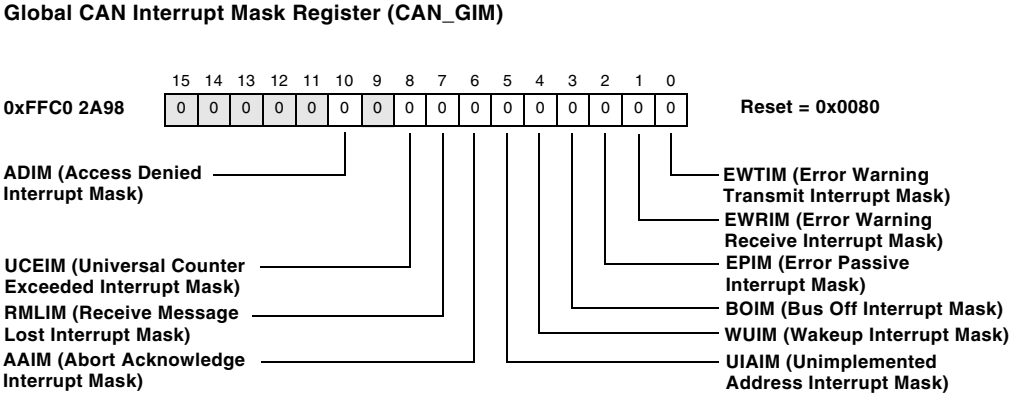


Figure 19-18. Global CAN Interrupt Mask Register

CAN_GIS Global CAN Interrupt Status Register

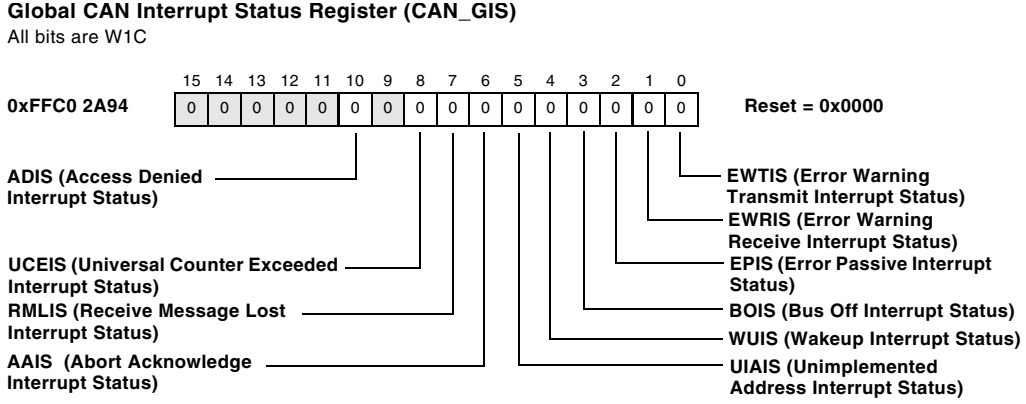


Figure 19-19. Global CAN Interrupt Status Register

CAN_GIF Global CAN Interrupt Flag Register

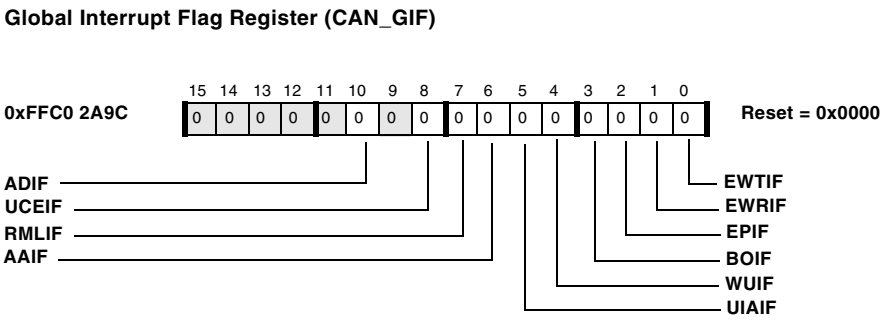


Figure 19-20. Global Interrupt Flag Register

Mailbox/Mask Registers

Figure 19-21 through Figure 19-30 on page 19-69 show the CAN mailbox and mask registers.

CAN_AMxx Mailbox Acceptance Registers

The value of the acceptance mask register does not matter when the AME bit is zero. If AME is set, only those bits that have the corresponding mask bit cleared are compared to the received message ID. A bit position that is one in the mask register does not need to match.

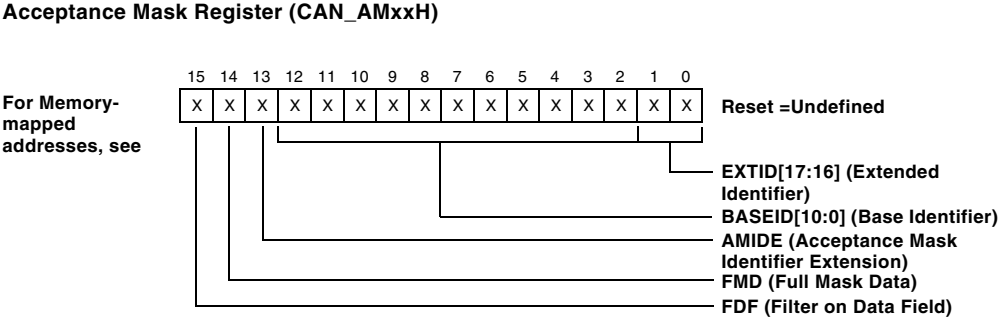


Figure 19-21. Acceptance Mask Register (H)

Table 19-10. Acceptance Mask Register (H) Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_AM00H	0xFFC0 2B04
CAN_AM01H	0xFFC0 2B0C
CAN_AM02H	0xFFC0 2B14
CAN_AM03H	0xFFC0 2B1C
CAN_AM04H	0xFFC0 2B24
CAN_AM05H	0xFFC0 2B2C
CAN_AM06H	0xFFC0 2B34
CAN_AM07H	0xFFC0 2B3C
CAN_AM08H	0xFFC0 2B44
CAN_AM09H	0xFFC0 2B4C
CAN_AM10H	0xFFC0 2B54
CAN_AM11H	0xFFC0 2B5C
CAN_AM12H	0xFFC0 2B64
CAN_AM13H	0xFFC0 2B6C
CAN_AM14H	0xFFC0 2B74

Table 19-10. Acceptance Mask Register (H) Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_AM15H	0xFFC0 2B7C
CAN_AM16H	0xFFC0 2B84
CAN_AM17H	0xFFC0 2B8C
CAN_AM18H	0xFFC0 2B94
CAN_AM19H	0xFFC0 2B9C
CAN_AM20H	0xFFC0 2BA4
CAN_AM21H	0xFFC0 2BAC
CAN_AM22H	0xFFC0 2BB4
CAN_AM23H	0xFFC0 2BBC
CAN_AM24H	0xFFC0 2BC4
CAN_AM25H	0xFFC0 2BCC
CAN_AM26H	0xFFC0 2BD4
CAN_AM27H	0xFFC0 2BDC
CAN_AM28H	0xFFC0 2BE4
CAN_AM29H	0xFFC0 2BEC
CAN_AM30H	0xFFC0 2BF4
CAN_AM31H	0xFFC0 2BFC

Acceptance Mask Register (CAN_AMxxL)

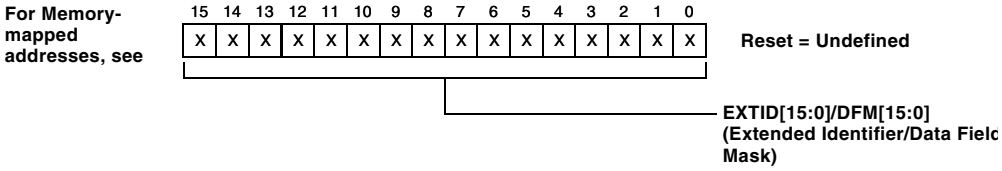


Figure 19-22. Acceptance Mask Register (L)

Table 19-11. Acceptance Mask Register (L) Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_AM00L	0xFFC0 2B00
CAN_AM01L	0xFFC0 2B08
CAN_AM02L	0xFFC0 2B10
CAN_AM03L	0xFFC0 2B18
CAN_AM04L	0xFFC0 2B20
CAN_AM05L	0xFFC0 2B28
CAN_AM06L	0xFFC0 2B30
CAN_AM07L	0xFFC0 2B38
CAN_AM08L	0xFFC0 2B40
CAN_AM09L	0xFFC0 2B48
CAN_AM10L	0xFFC0 2B50
CAN_AM11L	0xFFC0 2B58
CAN_AM12L	0xFFC0 2B60
CAN_AM13L	0xFFC0 2B68

Table 19-11. Acceptance Mask Register (L) Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_AM14L	0xFFC0 2B70
CAN_AM15L	0xFFC0 2B78
CAN_AM16L	0xFFC0 2B80
CAN_AM17L	0xFFC0 2B88
CAN_AM18L	0xFFC0 2B90
CAN_AM19L	0xFFC0 2B98
CAN_AM20L	0xFFC0 2BA0
CAN_AM21L	0xFFC0 2BA8
CAN_AM22L	0xFFC0 2BB0
CAN_AM23L	0xFFC0 2BB8
CAN_AM24L	0xFFC0 2BC0
CAN_AM25L	0xFFC0 2BC8
CAN_AM26L	0xFFC0 2BD0
CAN_AM27L	0xFFC0 2BD8
CAN_AM28L	0xFFC0 2BE0
CAN_AM29L	0xFFC0 2BE8
CAN_AM30L	0xFFC0 2BF0
CAN_AM31L	0xFFC0 2BF8

CAN_MBxx_ID1 Registers

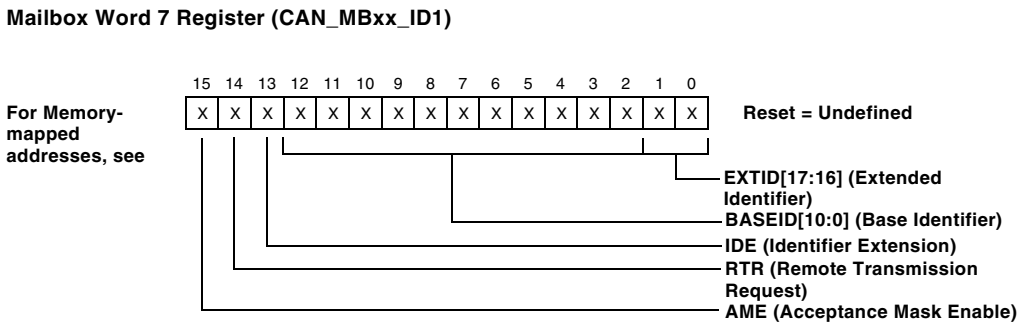


Figure 19-23. Mailbox Word 7 Register

Table 19-12. Mailbox Word 7 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_ID1	0xFFC0 2C1C
CAN_MB01_ID1	0xFFC0 2C3C
CAN_MB02_ID1	0xFFC0 2C5C
CAN_MB03_ID1	0xFFC0 2C7C
CAN_MB04_ID1	0xFFC0 2C9C
CAN_MB05_ID1	0xFFC0 2CBC
CAN_MB06_ID1	0xFFC0 2CDC
CAN_MB07_ID1	0xFFC0 2CFC
CAN_MB08_ID1	0xFFC0 2D1C
CAN_MB09_ID1	0xFFC0 2D3C
CAN_MB10_ID1	0xFFC0 2D5C
CAN_MB11_ID1	0xFFC0 2D7C
CAN_MB12_ID1	0xFFC0 2D9C
CAN_MB13_ID1	0xFFC0 2DBC

Table 19-12. Mailbox Word 7 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB14_ID1	0xFFC0 2DDC
CAN_MB15_ID1	0xFFC0 2DFC
CAN_MB16_ID1	0xFFC0 2E1C
CAN_MB17_ID1	0xFFC0 2E3C
CAN_MB18_ID1	0xFFC0 2E5C
CAN_MB19_ID1	0xFFC0 2E7C
CAN_MB20_ID1	0xFFC0 2E9C
CAN_MB21_ID1	0xFFC0 2EBC
CAN_MB22_ID1	0xFFC0 2EDC
CAN_MB23_ID1	0xFFC0 2EFC
CAN_MB24_ID1	0xFFC0 2F1C
CAN_MB25_ID1	0xFFC0 2F3C
CAN_MB26_ID1	0xFFC0 2F5C
CAN_MB27_ID1	0xFFC0 2F7C
CAN_MB28_ID1	0xFFC0 2F9C
CAN_MB29_ID1	0xFFC0 2FBC
CAN_MB30_ID1	0xFFC0 2FDC
CAN_MB31_ID1	0xFFC0 2FFC

CAN_MBxx_ID0 Registers

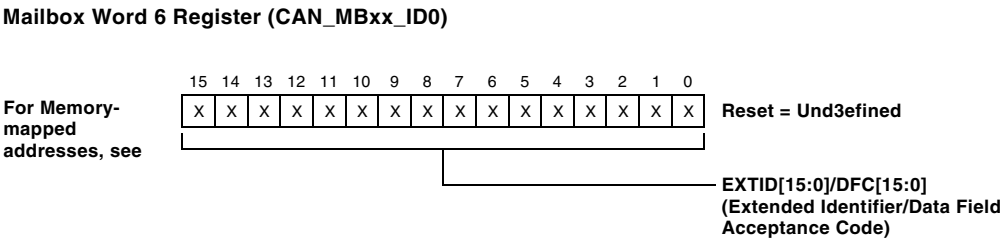


Figure 19-24. Mailbox Word 6 Register

Table 19-13. Mailbox Word 6 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_ID0	0xFFC0 2C18
CAN_MB01_ID0	0xFFC0 2C38
CAN_MB02_ID0	0xFFC0 2C58
CAN_MB03_ID0	0xFFC0 2C78
CAN_MB04_ID0	0xFFC0 2C98
CAN_MB05_ID0	0xFFC0 2CB8
CAN_MB06_ID0	0xFFC0 2CD8
CAN_MB07_ID0	0xFFC0 2CF8
CAN_MB08_ID0	0xFFC0 2D18
CAN_MB09_ID0	0xFFC0 2D38
CAN_MB10_ID0	0xFFC0 2D58
CAN_MB11_ID0	0xFFC0 2D78
CAN_MB12_ID0	0xFFC0 2D98
CAN_MB13_ID0	0xFFC0 2DB8
CAN_MB14_ID0	0xFFC0 2DD8
CAN_MB15_ID0	0xFFC0 2DF8

Table 19-13. Mailbox Word 6 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB16_ID0	0xFFC0 2E18
CAN_MB17_ID0	0xFFC0 2E38
CAN_MB18_ID0	0xFFC0 2E58
CAN_MB19_ID0	0xFFC0 2E78
CAN_MB20_ID0	0xFFC0 2E98
CAN_MB21_ID0	0xFFC0 2EB8
CAN_MB22_ID0	0xFFC0 2ED8
CAN_MB23_ID0	0xFFC0 2EF8
CAN_MB24_ID0	0xFFC0 2F18
CAN_MB25_ID0	0xFFC0 2F38
CAN_MB26_ID0	0xFFC0 2F58
CAN_MB27_ID0	0xFFC0 2F78
CAN_MB28_ID0	0xFFC0 2F98
CAN_MB29_ID0	0xFFC0 2FB8
CAN_MB30_ID0	0xFFC0 2FD8
CAN_MB31_ID0	0xFFC0 2FF8

CAN_MBxx_TIMESTAMP Registers

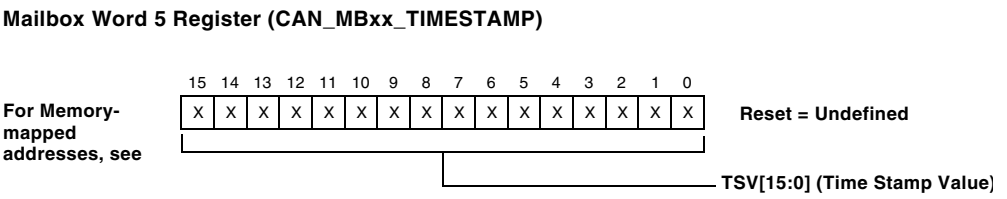


Figure 19-25. Mailbox Word 5 Register

Table 19-14. Mailbox Word 5 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_TIMESTAMP	0xFFC0 2C14
CAN_MB01_TIMESTAMP	0xFFC0 2C34
CAN_MB02_TIMESTAMP	0xFFC0 2C54
CAN_MB03_TIMESTAMP	0xFFC0 2C74
CAN_MB04_TIMESTAMP	0xFFC0 2C94
CAN_MB05_TIMESTAMP	0xFFC0 2CB4
CAN_MB06_TIMESTAMP	0xFFC0 2CD4
CAN_MB07_TIMESTAMP	0xFFC0 2CF4
CAN_MB08_TIMESTAMP	0xFFC0 2D14
CAN_MB09_TIMESTAMP	0xFFC0 2D34
CAN_MB10_TIMESTAMP	0xFFC0 2D54
CAN_MB11_TIMESTAMP	0xFFC0 2D74
CAN_MB12_TIMESTAMP	0xFFC0 2D94
CAN_MB13_TIMESTAMP	0xFFC0 2DB4
CAN_MB14_TIMESTAMP	0xFFC0 2DD4
CAN_MB15_TIMESTAMP	0xFFC0 2DF4
CAN_MB16_TIMESTAMP	0xFFC0 2E14

Table 19-14. Mailbox Word 5 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB17_TIMESTAMP	0xFFC0 2E34
CAN_MB18_TIMESTAMP	0xFFC0 2E54
CAN_MB19_TIMESTAMP	0xFFC0 2E74
CAN_MB20_TIMESTAMP	0xFFC0 2E94
CAN_MB21_TIMESTAMP	0xFFC0 2EB4
CAN_MB22_TIMESTAMP	0xFFC0 2ED4
CAN_MB23_TIMESTAMP	0xFFC0 2EF4
CAN_MB24_TIMESTAMP	0xFFC0 2F14
CAN_MB25_TIMESTAMP	0xFFC0 2F34
CAN_MB26_TIMESTAMP	0xFFC0 2F54
CAN_MB27_TIMESTAMP	0xFFC0 2F74
CAN_MB28_TIMESTAMP	0xFFC0 2F94
CAN_MB29_TIMESTAMP	0xFFC0 2FB4
CAN_MB30_TIMESTAMP	0xFFC0 2FD4
CAN_MB31_TIMESTAMP	0xFFC0 2FF4

CAN_MBxx_LENGTH Registers

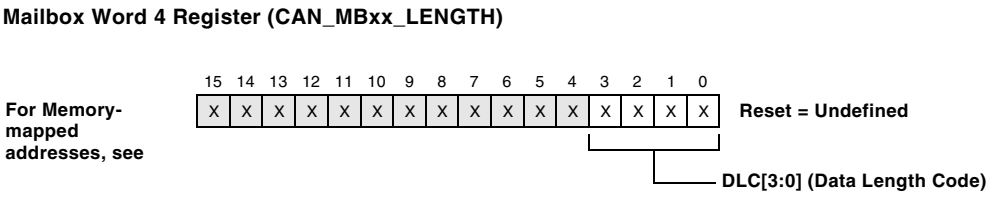


Figure 19-26. Mailbox Word 4 Register

Table 19-15. Mailbox Word 4 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_LENGTH	0xFFC0 2C10
CAN_MB01_LENGTH	0xFFC0 2C30
CAN_MB02_LENGTH	0xFFC0 2C50
CAN_MB03_LENGTH	0xFFC0 2C70
CAN_MB04_LENGTH	0xFFC0 2C90
CAN_MB05_LENGTH	0xFFC0 2CB0
CAN_MB06_LENGTH	0xFFC0 2CD0
CAN_MB07_LENGTH	0xFFC0 2CF0
CAN_MB08_LENGTH	0xFFC0 2D10
CAN_MB09_LENGTH	0xFFC0 2D30
CAN_MB10_LENGTH	0xFFC0 2D50
CAN_MB11_LENGTH	0xFFC0 2D70
CAN_MB12_LENGTH	0xFFC0 2D90
CAN_MB13_LENGTH	0xFFC0 2DB0
CAN_MB14_LENGTH	0xFFC0 2DD0
CAN_MB15_LENGTH	0xFFC0 2DF0
CAN_MB16_LENGTH	0xFFC0 2E10

Table 19-15. Mailbox Word 4 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB17_LENGTH	0xFFC0 2E30
CAN_MB18_LENGTH	0xFFC0 2E50
CAN_MB19_LENGTH	0xFFC0 2E70
CAN_MB20_LENGTH	0xFFC0 2E90
CAN_MB21_LENGTH	0xFFC0 2EB0
CAN_MB22_LENGTH	0xFFC0 2ED0
CAN_MB23_LENGTH	0xFFC0 2EF0
CAN_MB24_LENGTH	0xFFC0 2F10
CAN_MB25_LENGTH	0xFFC0 2F30
CAN_MB26_LENGTH	0xFFC0 2F50
CAN_MB27_LENGTH	0xFFC0 2F70
CAN_MB28_LENGTH	0xFFC0 2F90
CAN_MB29_LENGTH	0xFFC0 2FB0
CAN_MB30_LENGTH	0xFFC0 2FD0
CAN_MB31_LENGTH	0xFFC0 2FF0

CAN_MBxx_DATAx Registers

The following are the descriptions of Mailbox Word registers (CAN_MBxx_DATA3/2/1/0) and their appropriate memory-mapped addresses.

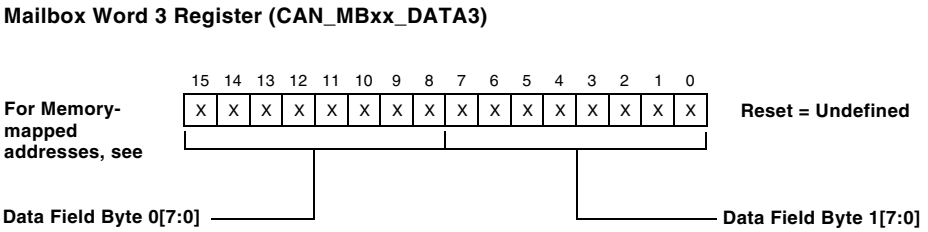


Figure 19-27. Mailbox Word 3 Register

Table 19-16. Mailbox Word 3 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA3	0xFFC0 2C0C
CAN_MB01_DATA3	0xFFC0 2C2C
CAN_MB02_DATA3	0xFFC0 2C4C
CAN_MB03_DATA3	0xFFC0 2C6C
CAN_MB04_DATA3	0xFFC0 2C8C
CAN_MB05_DATA3	0xFFC0 2CAC
CAN_MB06_DATA3	0xFFC0 2CCC
CAN_MB07_DATA3	0xFFC0 2CEC
CAN_MB08_DATA3	0xFFC0 2D0C
CAN_MB09_DATA3	0xFFC0 2D2C
CAN_MB10_DATA3	0xFFC0 2D4C
CAN_MB11_DATA3	0xFFC0 2D6C
CAN_MB12_DATA3	0xFFC0 2D8C

Table 19-16. Mailbox Word 3 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB13_DATA3	0xFFC0 2DAC
CAN_MB14_DATA3	0xFFC0 2DCC
CAN_MB15_DATA3	0xFFC0 2DEC
CAN_MB16_DATA3	0xFFC0 2E0C
CAN_MB17_DATA3	0xFFC0 2E2C
CAN_MB18_DATA3	0xFFC0 2E4C
CAN_MB19_DATA3	0xFFC0 2E6C
CAN_MB20_DATA3	0xFFC0 2E8C
CAN_MB21_DATA3	0xFFC0 2EAC
CAN_MB22_DATA3	0xFFC0 2ECC
CAN_MB23_DATA3	0xFFC0 2EEC
CAN_MB24_DATA3	0xFFC0 2F0C
CAN_MB25_DATA3	0xFFC0 2F2C
CAN_MB26_DATA3	0xFFC0 2F4C
CAN_MB27_DATA3	0xFFC0 2F6C
CAN_MB28_DATA3	0xFFC0 2F8C
CAN_MB29_DATA3	0xFFC0 2FAC
CAN_MB30_DATA3	0xFFC0 2FCC
CAN_MB31_DATA3	0xFFC0 2FEC

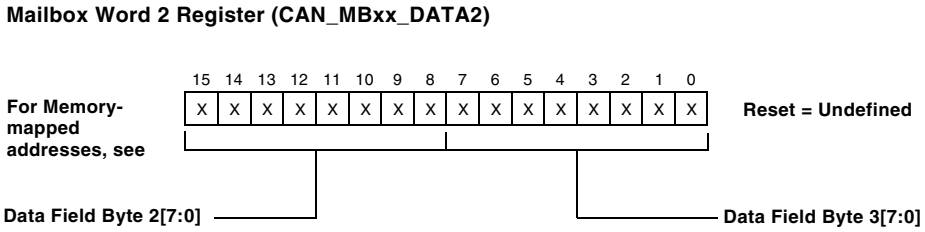


Figure 19-28. Mailbox Word 2 Register

Table 19-17. Mailbox Word 2 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA2	0xFFC0 2C08
CAN_MB01_DATA2	0xFFC0 2C28
CAN_MB02_DATA2	0xFFC0 2C48
CAN_MB03_DATA2	0xFFC0 2C68
CAN_MB04_DATA2	0xFFC0 2C88
CAN_MB05_DATA2	0xFFC0 2CA8
CAN_MB06_DATA2	0xFFC0 2CC8
CAN_MB07_DATA2	0xFFC0 2CE8
CAN_MB08_DATA2	0xFFC0 2D08
CAN_MB09_DATA2	0xFFC0 2D28
CAN_MB10_DATA2	0xFFC0 2D48
CAN_MB11_DATA2	0xFFC0 2D68
CAN_MB12_DATA2	0xFFC0 2D88
CAN_MB13_DATA2	0xFFC0 2DA8
CAN_MB14_DATA2	0xFFC0 2DC8
CAN_MB15_DATA2	0xFFC0 2DE8
CAN_MB16_DATA2	0xFFC0 2E08
CAN_MB17_DATA2	0xFFC0 2E28

Table 19-17. Mailbox Word 2 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB18_DATA2	0xFFC0 2E48
CAN_MB19_DATA2	0xFFC0 2E68
CAN_MB20_DATA2	0xFFC0 2E88
CAN_MB21_DATA2	0xFFC0 2EA8
CAN_MB22_DATA2	0xFFC0 2EC8
CAN_MB23_DATA2	0xFFC0 2EE8
CAN_MB24_DATA2	0xFFC0 2F08
CAN_MB25_DATA2	0xFFC0 2F28
CAN_MB26_DATA2	0xFFC0 2F48
CAN_MB27_DATA2	0xFFC0 2F68
CAN_MB28_DATA2	0xFFC0 2F88
CAN_MB29_DATA2	0xFFC0 2FA8
CAN_MB30_DATA2	0xFFC0 2FC8
CAN_MB31_DATA2	0xFFC0 2FE8

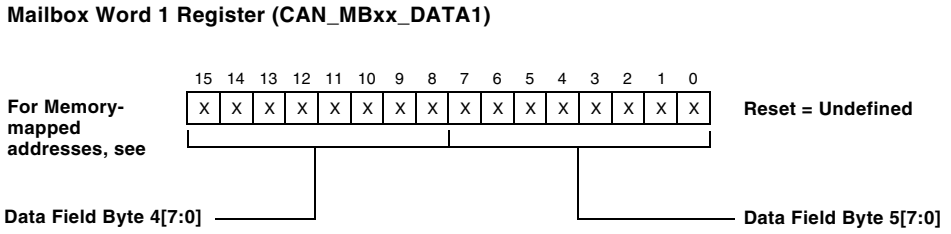


Figure 19-29. Mailbox Word 1 Register

Table 19-18. Mailbox Word 1 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA1	0xFFC0 2C04
CAN_MB01_DATA1	0xFFC0 2C24
CAN_MB02_DATA1	0xFFC0 2C44
CAN_MB03_DATA1	0xFFC0 2C64
CAN_MB04_DATA1	0xFFC0 2C84
CAN_MB05_DATA1	0xFFC0 2CA4
CAN_MB06_DATA1	0xFFC0 2CC4
CAN_MB07_DATA1	0xFFC0 2CE4
CAN_MB08_DATA1	0xFFC0 2D04
CAN_MB09_DATA1	0xFFC0 2D24
CAN_MB10_DATA1	0xFFC0 2D44
CAN_MB11_DATA1	0xFFC0 2D64
CAN_MB12_DATA1	0xFFC0 2D84
CAN_MB13_DATA1	0xFFC0 2DA4
CAN_MB14_DATA1	0xFFC0 2DC4
CAN_MB15_DATA1	0xFFC0 2DE4
CAN_MB16_DATA1	0xFFC0 2E04
CAN_MB17_DATA1	0xFFC0 2E24

Table 19-18. Mailbox Word 1 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB18_DATA1	0xFFC0 2E44
CAN_MB19_DATA1	0xFFC0 2E64
CAN_MB20_DATA1	0xFFC0 2E84
CAN_MB21_DATA1	0xFFC0 2EA4
CAN_MB22_DATA1	0xFFC0 2EC4
CAN_MB23_DATA1	0xFFC0 2EE4
CAN_MB24_DATA1	0xFFC0 2F04
CAN_MB25_DATA1	0xFFC0 2F24
CAN_MB26_DATA1	0xFFC0 2F44
CAN_MB27_DATA1	0xFFC0 2F64
CAN_MB28_DATA1	0xFFC0 2F84
CAN_MB29_DATA1	0xFFC0 2FA4
CAN_MB30_DATA1	0xFFC0 2FC4
CAN_MB31_DATA1	0xFFC0 2FE4

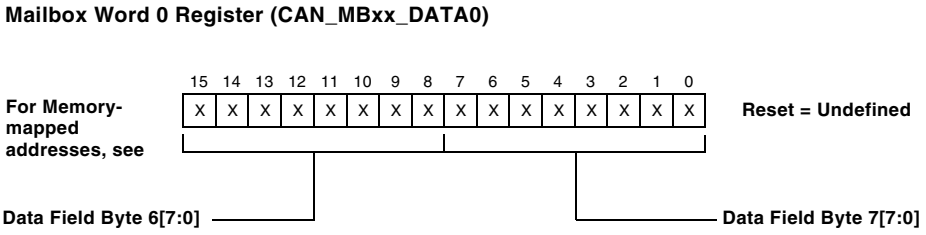


Figure 19-30. Mailbox Word 0 Register

Table 19-19. Mailbox Word 0 Register Memory-Mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA0	0xFFC0 2C00
CAN_MB01_DATA0	0xFFC0 2C20
CAN_MB02_DATA0	0xFFC0 2C40
CAN_MB03_DATA0	0xFFC0 2C60
CAN_MB04_DATA0	0xFFC0 2C80
CAN_MB05_DATA0	0xFFC0 2CA0
CAN_MB06_DATA0	0xFFC0 2CC0
CAN_MB07_DATA0	0xFFC0 2CE0
CAN_MB08_DATA0	0xFFC0 2D00
CAN_MB09_DATA0	0xFFC0 2D20
CAN_MB10_DATA0	0xFFC0 2D40
CAN_MB11_DATA0	0xFFC0 2D60
CAN_MB12_DATA0	0xFFC0 2D80
CAN_MB13_DATA0	0xFFC0 2DA0
CAN_MB14_DATA0	0xFFC0 2DC0
CAN_MB15_DATA0	0xFFC0 2DE0
CAN_MB16_DATA0	0xFFC0 2E00
CAN_MB17_DATA0	0xFFC0 2E20

Table 19-19. Mailbox Word 0 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB18_DATA0	0xFFC0 2E40
CAN_MB19_DATA0	0xFFC0 2E60
CAN_MB20_DATA0	0xFFC0 2E80
CAN_MB21_DATA0	0xFFC0 2EA0
CAN_MB22_DATA0	0xFFC0 2EC0
CAN_MB23_DATA0	0xFFC0 2EE0
CAN_MB24_DATA0	0xFFC0 2F00
CAN_MB25_DATA0	0xFFC0 2F20
CAN_MB26_DATA0	0xFFC0 2F40
CAN_MB27_DATA0	0xFFC0 2F60
CAN_MB28_DATA0	0xFFC0 2F80
CAN_MB29_DATA0	0xFFC0 2FA0
CAN_MB30_DATA0	0xFFC0 2FC0
CAN_MB31_DATA0	0xFFC0 2FE0

Mailbox Control Registers

Figure 19-31 through Figure 19-57 on page 19-84 show the mailbox control registers.

CAN_MCx Mailbox Configuration Registers

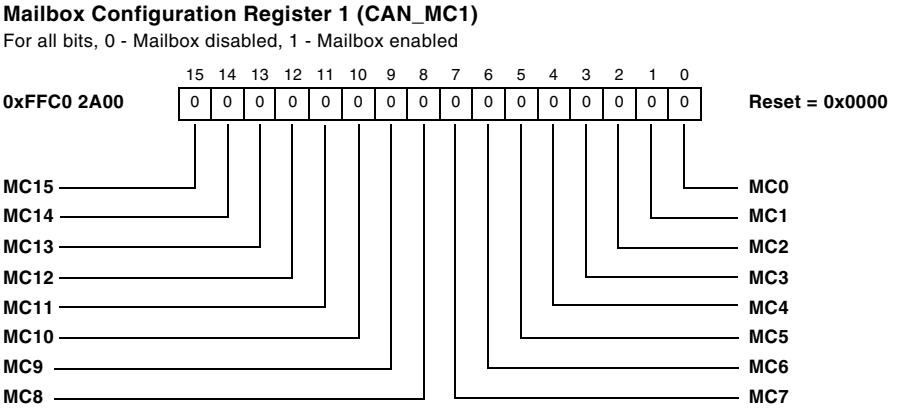


Figure 19-31. Mailbox Configuration Register 1

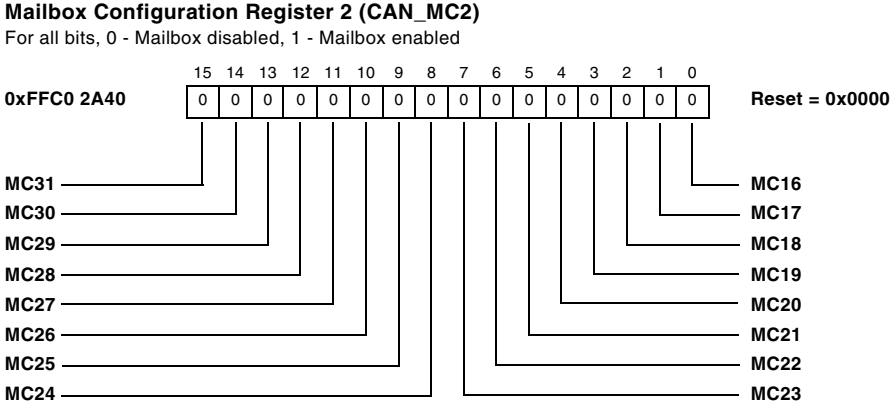


Figure 19-32. Mailbox Configuration Register 2

CAN_MDx Mailbox Direction Registers

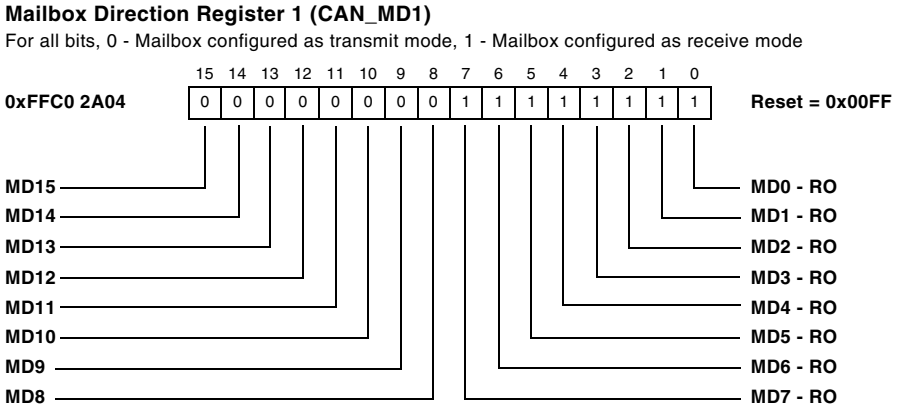


Figure 19-33. Mailbox Direction Register 1

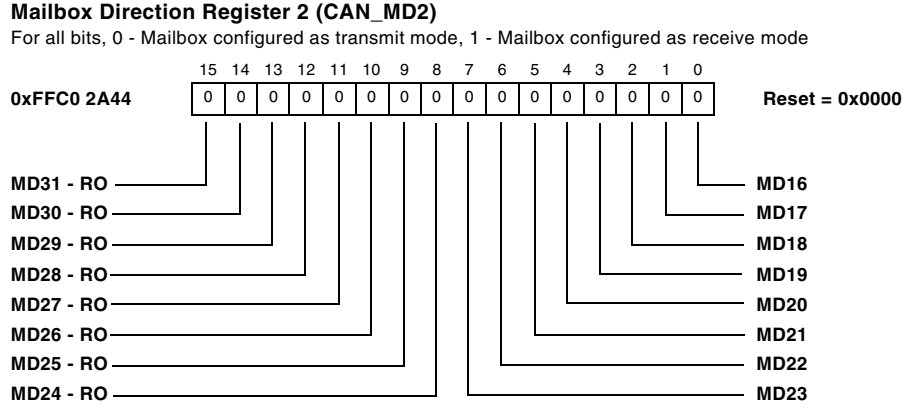


Figure 19-34. Mailbox Direction Register 2

CAN_RMPx Registers

Receive Message Pending Register 1 (CAN_RMP1)
All bits are W1C

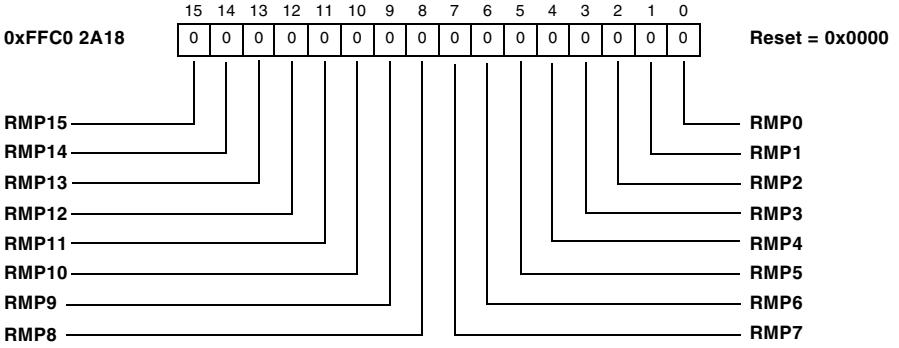


Figure 19-35. Receive Message Pending Register 1

Receive Message Pending Register 2 (CAN_RMP2)
All bits are W1C

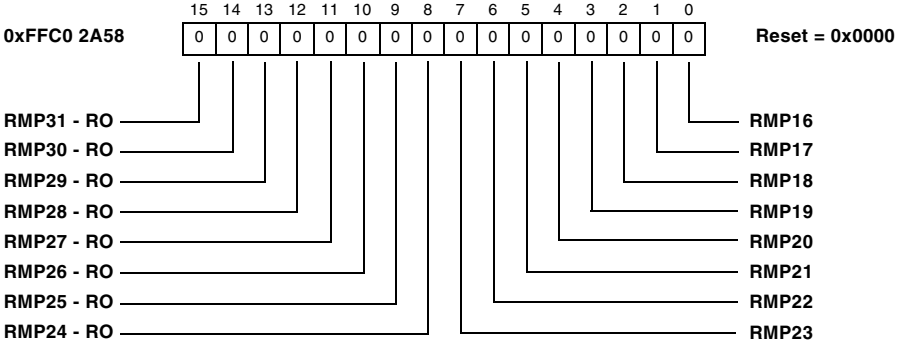


Figure 19-36. Receive Message Pending Register 2

CAN_RMLx Registers

Receive Message Lost Register 1 (CAN_RML1)
RO

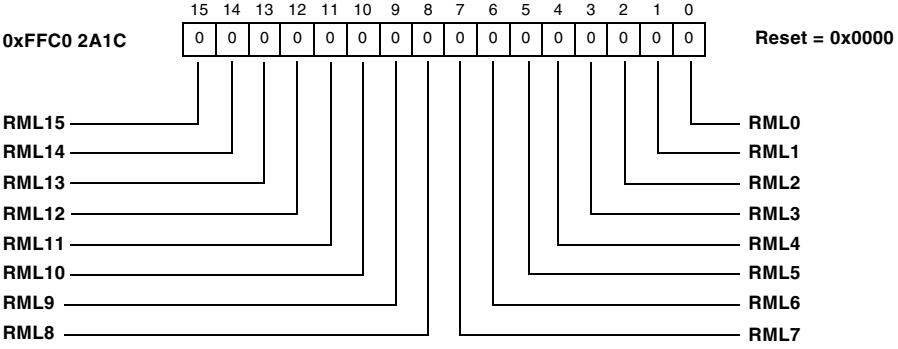


Figure 19-37. Receive Message Lost Register 1

Receive Message Lost Register 2 (CAN_RML2)
RO

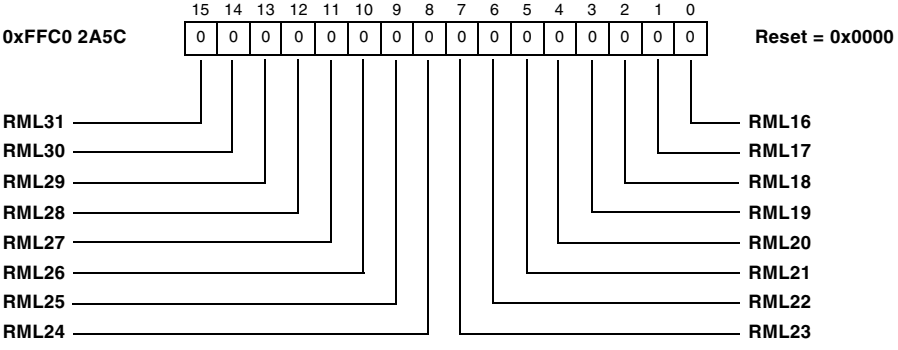


Figure 19-38. Receive Message Lost Register 2

CAN_OPSSx Register

Overwrite Protection/Single Shot Transmission Register 1 (CAN_OPSS1)

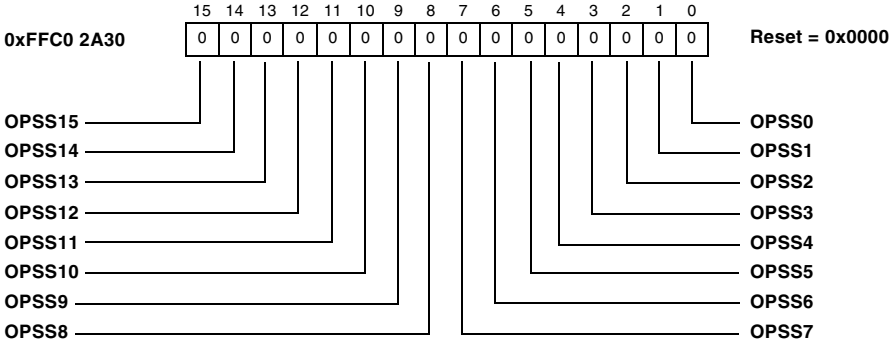


Figure 19-39. Overwrite Protection/Single Shot Transmission Register 1

Overwrite Protection/Single Shot Transmission Register 2 (CAN_OPSS2)

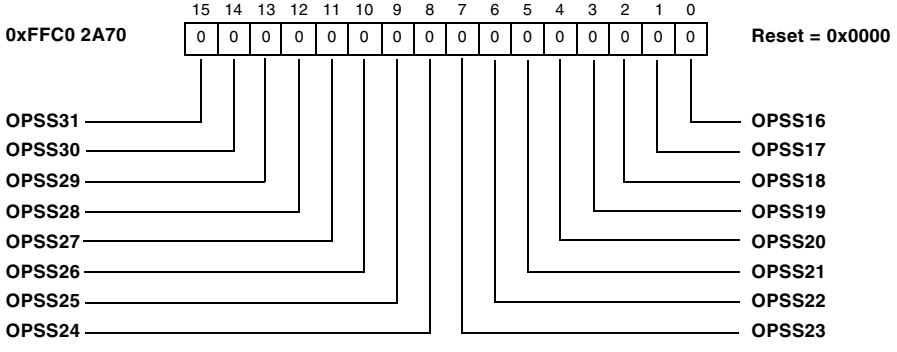


Figure 19-40. Overwrite Protection/Single Shot Transmission Register 2

CAN_TRSx Registers

Transmission Request Set Register 1 (CAN_TRS1)

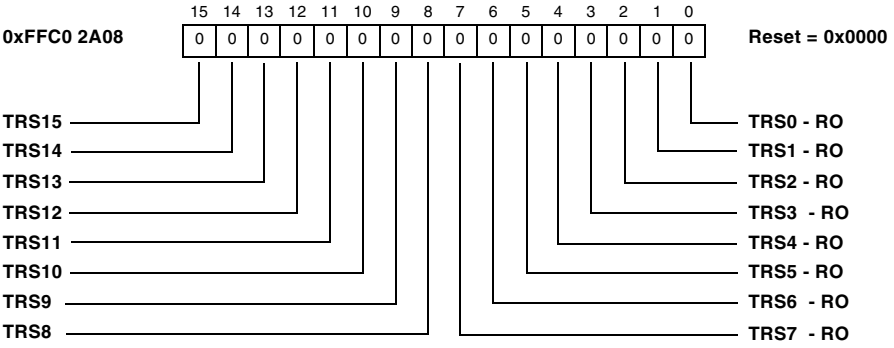


Figure 19-41. Transmission Request Set Register 1

Transmission Request Set Register 2 (CAN_TRS2)

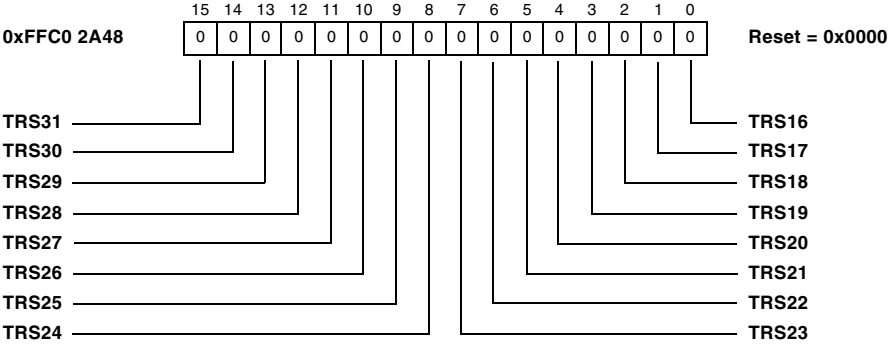


Figure 19-42. Transmission Request Set Register 2

CAN_TRRx Registers

Transmission Request Reset Register 1 (CAN_TRR1)

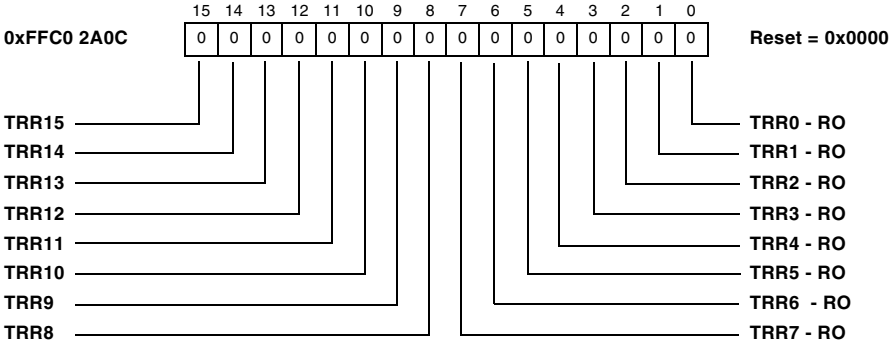


Figure 19-43. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CAN_TRR2)

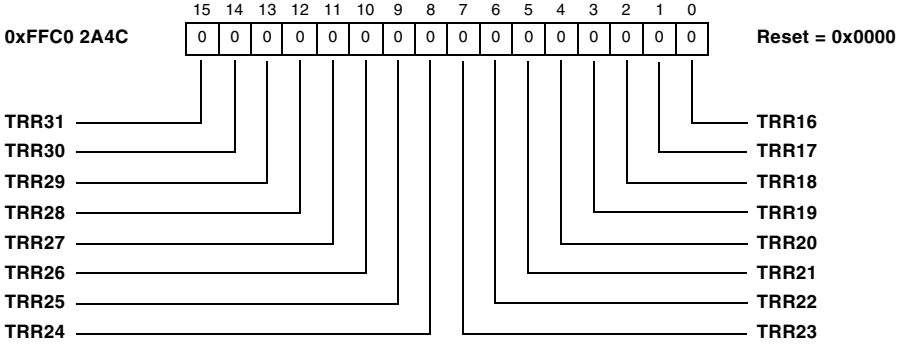


Figure 19-44. Transmission Request Reset Register 2

CAN_AAx Registers

Abort Acknowledge Register 1 (CAN_AA1)
All bits are W1C

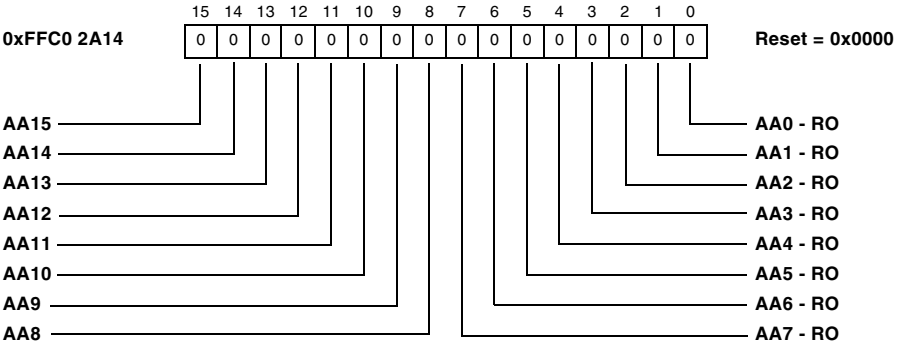


Figure 19-45. Abort Acknowledge Register 1

Abort Acknowledge Register 2 (CAN_AA2)
All bits are W1C

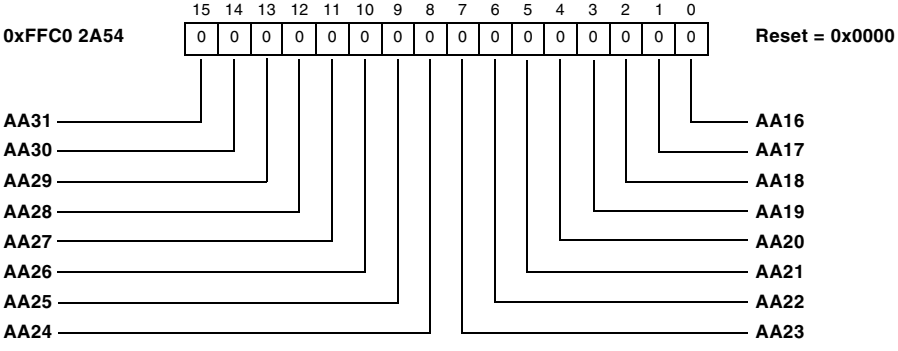


Figure 19-46. Abort Acknowledge Register 2

CAN_TAx Registers

Transmission Acknowledge Register 1 (CAN_TA1)
All bits are W1C

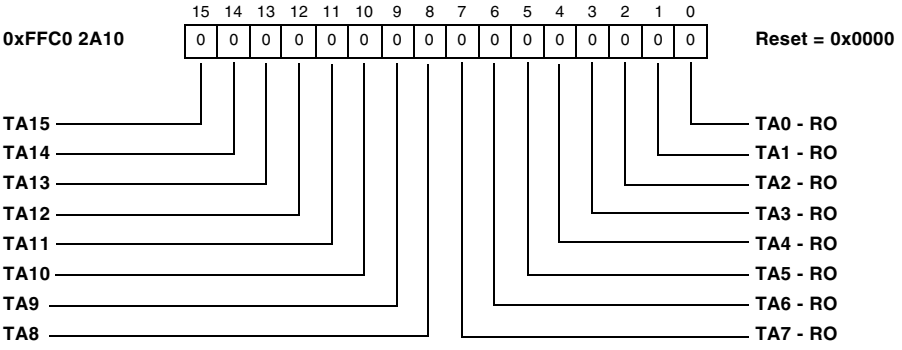


Figure 19-47. Transmission Acknowledge Register 1

Transmission Acknowledge Register 2 (CAN_TA2)
All bits are W1C

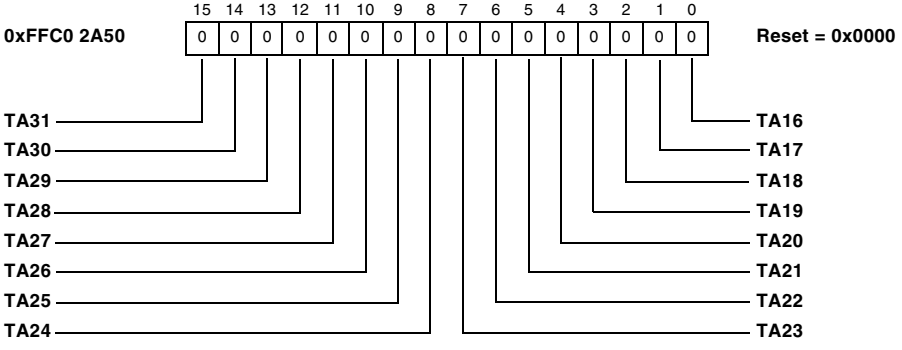


Figure 19-48. Transmission Acknowledge Register 2

CAN_MBTD Register

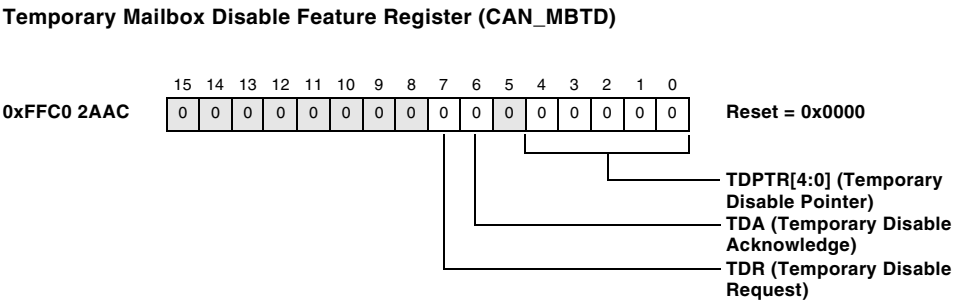


Figure 19-49. Temporary Mailbox Disable Register

CAN_RFHx Registers

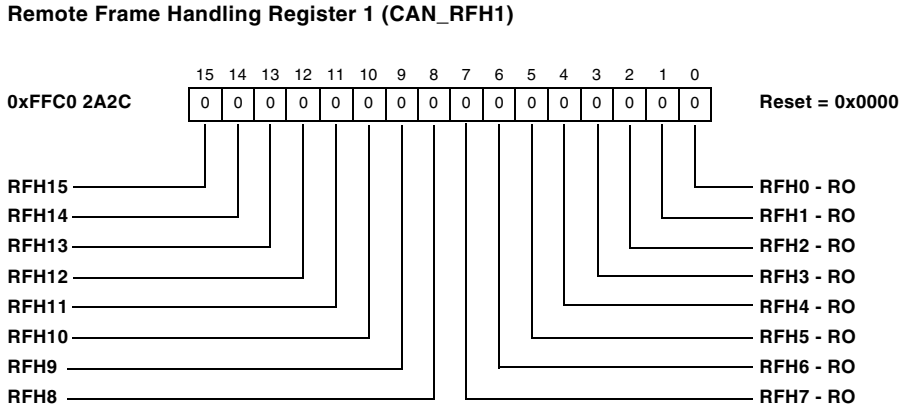


Figure 19-50. Remote Frame Handling Register 1

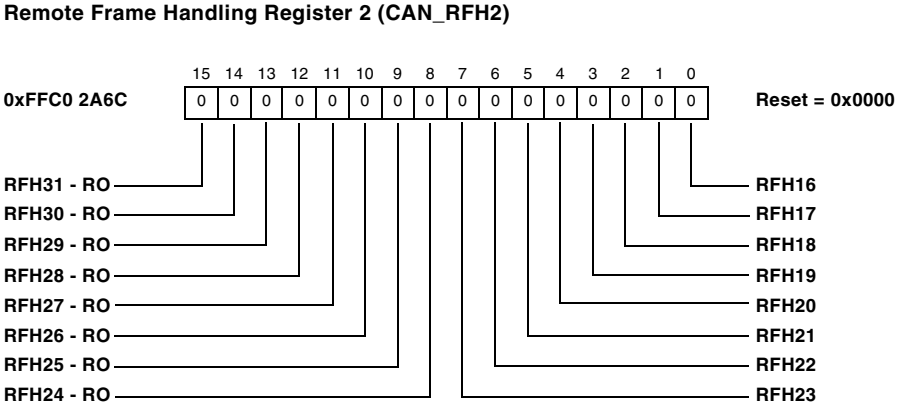


Figure 19-51. Remote Frame Handling Register 2

CAN_MBIMx Registers

Mailbox Interrupt Mask Register 1 (CAN_MBIM1)

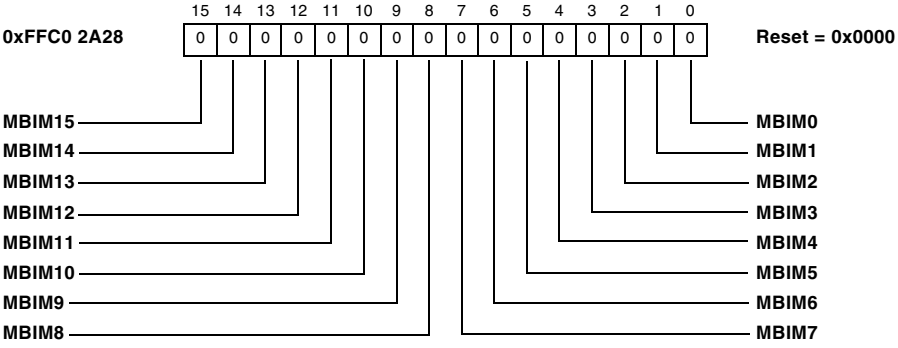


Figure 19-52. Mailbox Interrupt Mask Register 1

Mailbox Interrupt Mask Register 2 (CAN_MBIM2)

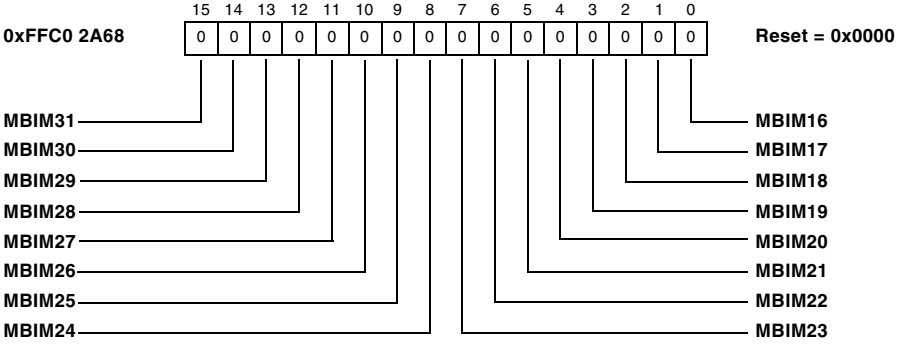


Figure 19-53. Mailbox Interrupt Mask Register 2

CAN_MBTIFx Registers

Mailbox Transmit Interrupt Flag Register 1 (CAN_MBTIF1)
All bits are W1C

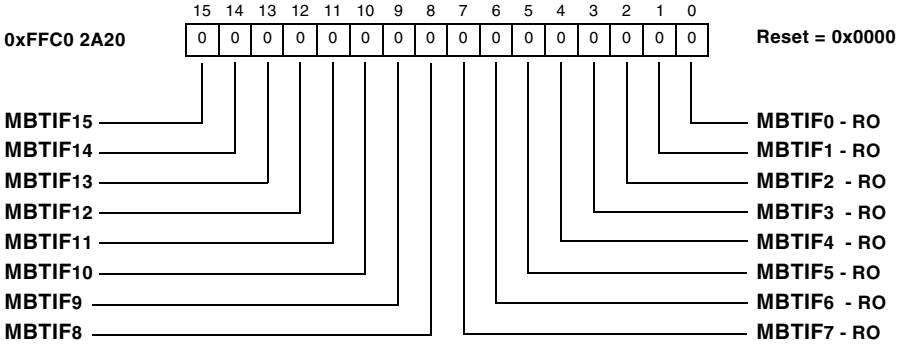


Figure 19-54. Mailbox Transmit Interrupt Flag Register 1

Mailbox Transmit Interrupt Flag Register 2 (CAN_MBTIF2)
All bits are W1C

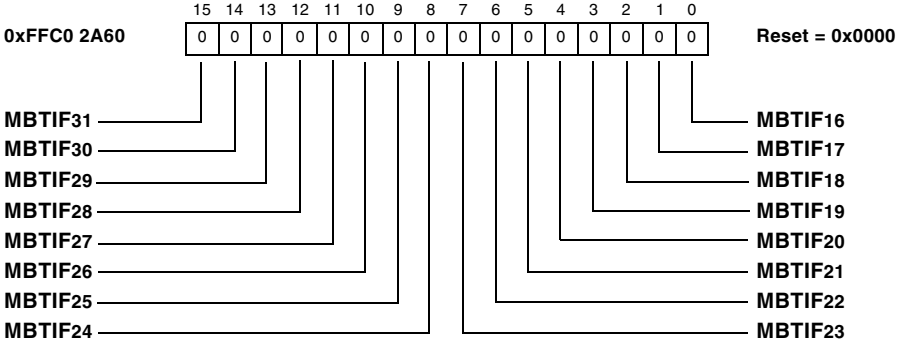


Figure 19-55. Mailbox Transmit Interrupt Flag Register 2

CAN_MBRIFx Registers

Mailbox Receive Interrupt Flag Register 1 (CAN_MBRIF1)
All bits are W1C

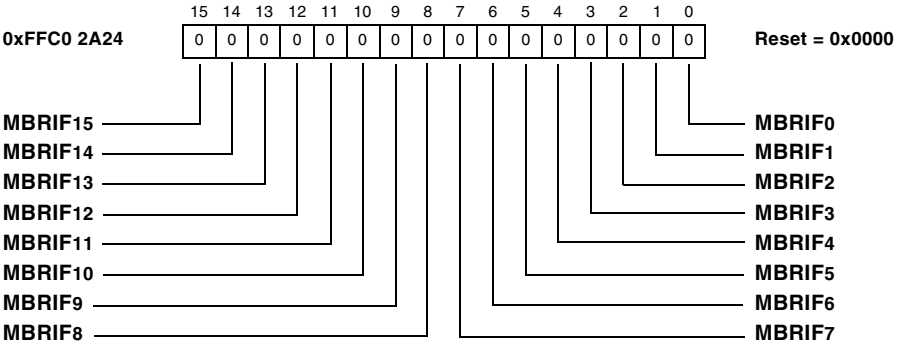


Figure 19-56. Mailbox Receive Interrupt Flag Register 1

Mailbox Receive Interrupt Flag Register 2 (CAN_MBRIF2)
All bits are W1C

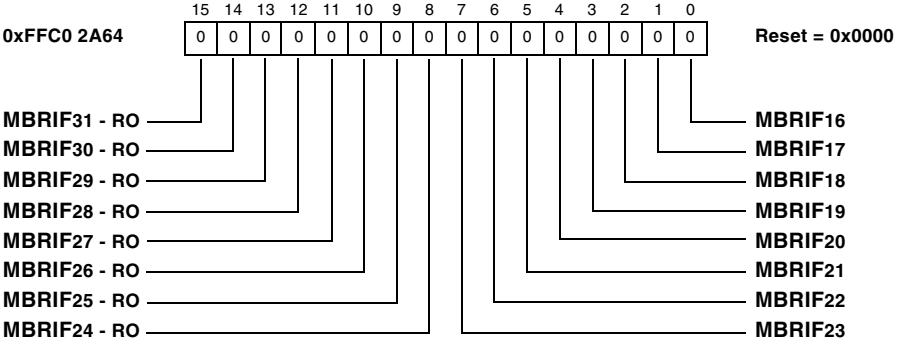


Figure 19-57. Mailbox Receive Interrupt Flag Register 2

Universal Counter Registers

Figure 19-58 through Figure 19-60 show the universal counter registers.

CAN_UCCNF Register

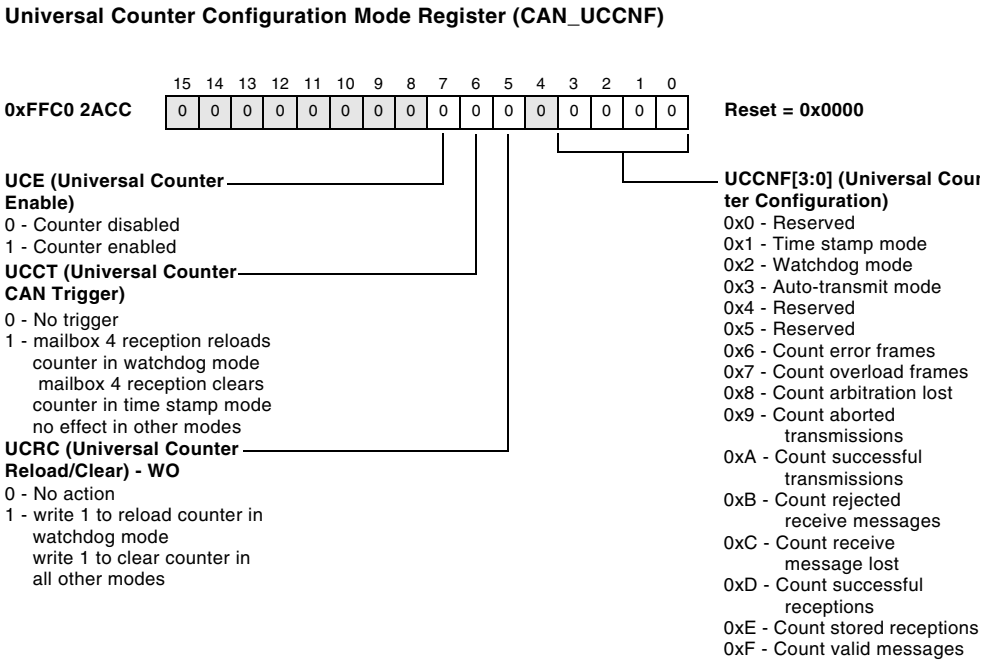


Figure 19-58. Universal Counter Configuration Mode Register

CAN_UCCNT Register

Universal Counter Register (CAN_UCCNT)

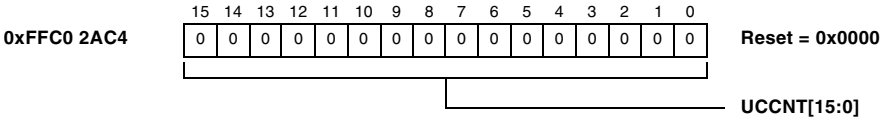


Figure 19-59. Universal Counter Register

CAN_UCRC Register

Universal Counter Reload/Capture Register (CAN_UCRC)

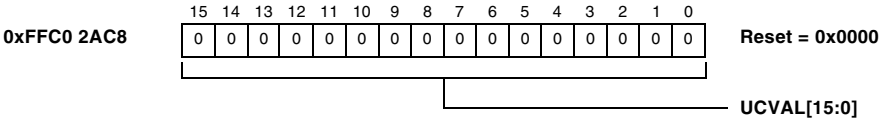


Figure 19-60. Universal Counter Reload/Capture Register

Error Registers

Figure 19-61 through Figure 19-63 show the CAN controller error registers.

CAN_CEC Register

CAN Error Counter Register (CAN_CEC)

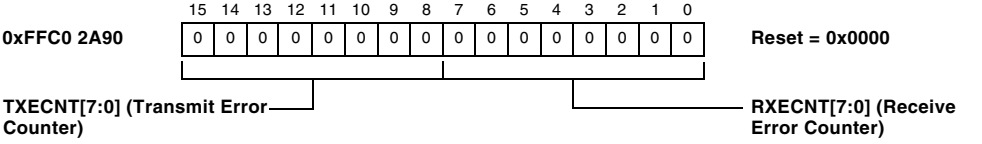


Figure 19-61. Error Counter Register

CAN_ESR Register

Error Status Register (CAN_ESR)
All bits are W1C

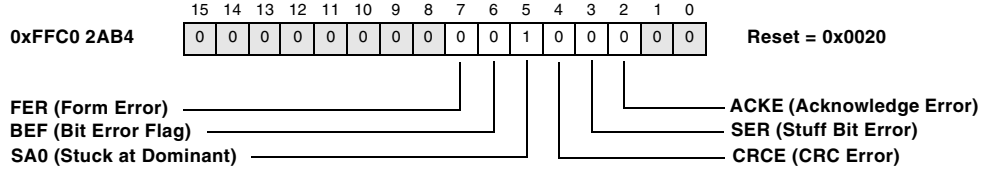


Figure 19-62. Error Status Register

CAN_EWR Register

CAN Error Counter Warning Level Register (CAN_EWR)

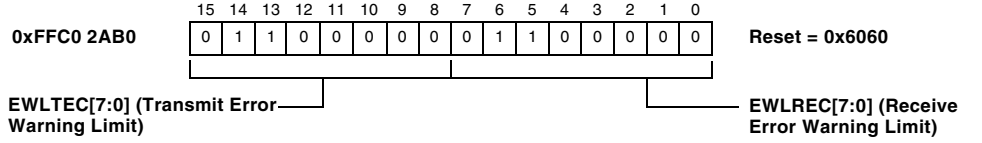


Figure 19-63. Error Counter Warning Level Register

Programming Examples

The following CAN code examples (Listing 19-2 through Listing 19-4 on page 19-92) 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 <defBF539.h>` for ADSP-BF539 projects).

CAN Setup Code

The following code initializes the port pins to connect to the CAN controller and configures the CAN timing parameters.

Listing 19-2. Initializing CAN

```
Initialize_CAN:
  /* =================================================
  ** Set CAN Bit Timing
  **
  ** CAN_TIMING - SJW, TSEG2, and TSEG1 governed by:
  ** SJW <= TSEG2 <= TSEG1
  **
  ** =================================================
  */
  P0.H = HI(CAN_TIMING);
  P0.L = LO(CAN_TIMING);
```

```
R0 = 0x0334(Z);  /* SJW = 3, TSEG2 = 3, TSEG1 = 4 */
W[P0] = R0;
SSYNC;

/* ================================================
** CAN_CLOCK  - Calculate Prescaler (BRP)
**
** Assume a 500kbps CAN rate is desired, which means
** the duration of the bit on the CAN bus (tBIT) is
** 2us. Using the tBIT formula from the HRM, solve for
** TQ:
**
** tBIT = TQ x (1 + (TSEG1 + 1) + (TSEG2 + 1))
** 2us  = TQ x (1 + (4 + 1) + (3 + 1))
** 2e-6 = TQ x (1 + 5 + 4)
** TQ = 2e-6 / 10
** TQ = 2e-7
**
** Once time quantum (TQ) is known, BRP can be derived
** from the TQ formula in the HRM. Assume the default
** PLL settings are used for the ADSP-BF538 EZ-KIT,
** which implies that System Clock (SCLK) is 50MHz:
**
** TQ = (BRP+1) / SCLK
** 2e-7 = (BRP+1) / 50e6
** (BRP+1) = 10
** BRP = 9
*/
P0.L = LO(CAN_CLOCK);
R0   = 9(Z);
W[P0] = R0;
SSYNC;

RTS;
```

Initializing and Enabling CAN Mailboxes

Before the CAN can transfer data, the mailbox area must be properly set up and the controller must be initialized properly.

Listing 19-3. Initializing and Enabling Mailboxes

```
CAN_Initialize_Mailboxes:
  P0.H = HI(CAN_MD1);  /* Configure Mailbox Direction */
  P0.L = LO(CAN_MD1);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(MD8));  /* Set MB08 for Transmit */
  BITSET(R0, BITPOS(MD9));  /* Set MB09 for Receive */
  W[P0] = R0;
  SSYNC;

  /* ================================================
  ** Populate CAN Mailbox Area
  **
  ** Mailbox 8 transmits ID 0x411 with 4 bytes of data
  ** Bytes 0 and 1 are a data pattern 0xAABB. Bytes 2
  ** and 3 will be a count value for the number of times
  ** that message is properly sent.
  **
  ** Mailbox 9 will receive message ID 0x007
  **
  ** ================================================
  */

  /* Initialize Mailbox 8 For Transmit */
  R0 = 0x411 << 2;  /* Put Message ID in correct slot */
  P0.L = LO(CAN_MB_ID1(8));  /* Access MB08 ID1 Register */
  W[P0] = R0;  /* Remote frame disabled, 11 bit ID */
```

```
R0 = 0;
P0.L = LO(CAN_MB_ID0(8));
W[P0] = R0;  /* Zero Out Lower ID Register */

R0 = 4;
P0.L = LO(CAN_MB_LENGTH(8));
W[P0] = R0;  /* Set DLC to 4 Bytes */

R0 = 0xAABB(Z);
P0.L = LO(CAN_MB_DATA3(8));
W[P0] = R0;  /* Byte0 = 0xAA, Byte1 = 0xBB */

R0 = 1;
P0.L = LO(CAN_MB_DATA2(8));
W[P0] = R0;  /* Initialize Count to 1 */

/* Initialize Mailbox 9 For Receive */
R0 = 0x007 << 2;  /* Put Message ID in correct slot */
P0.L = LO(CAN_MB_ID1(9));  /* Access MB08 ID1 Register */
W[P0] = R0;  /* Remote frame disabled, 11 bit ID */

R0 = 0;
P0.L = LO(CAN_MB_ID0(9));
W[P0] = R0;  /* Zero Out Lower ID Register */
SSYNC;

/* Enable the Configured Mailboxes */
P0.L = LO(CAN_MC1);
R0 = W[P0](Z);
BITSET(R0, BITPOS(MC8));  /* Enable MB08 */
BITSET(R0, BITPOS(MC9));  /* Enable MB09 */
W[P0] = R0;
SSYNC;
RTS;
```

Initiating CAN Transfers and Processing Interrupts

After the mailboxes are properly set up, transfers can be requested in the CAN controller. This code example initializes the CAN-level interrupts, takes the CAN controller out of configuration mode, requests a transfer, and then waits for and processes CAN TX and RX interrupts. This example assumes that the `CAN_RX_HANDLER` and `CAN_TX_HANDLER` have been properly registered in the system interrupt controller and that the interrupts are enabled properly in the `SIC_IMASK1` register.

Listing 19-4. CAN Transfers and Interrupts

```
CAN_SetupIRQs_and_Transfer:
  P0.H = HI(CAN_MBIM1);
  P0.L = LO(CAN_MBIM1);
  R0 = 0;
  BITSET(R0, BITPOS(MBIM8));  /* Enable Mailbox Interrupts */
  BITSET(R0, BITPOS(MBIM9));  /* for Mailboxes 8 and 9 */
  W[P0] = R0;
  SSYNC;

  /* Leave CAN Configuration Mode (Clear CCR) */
  P0.L = LO(CAN_CONTROL);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(CCR));
  W[P0] = R0;

  P0.L = LO(CAN_STATUS);
/* Wait for CAN Configuration Acknowledge (CCA) */
  WAIT_FOR_CCA_TO_CLEAR:
    R1 = W[P0](Z);
    CC = BITTST (R1, BITPOS(CCA));
    IF CC JUMP WAIT_FOR_CCA_TO_CLEAR;
  P0.L = LO(CAN_TRS1);
```

```
  R0 = TRS8;  /* Transmit Request MB08 */
  W[P0] = R0;  /* Issue Transmit Request */
  SSYNC;

  Wait_Here_For_IRQs:
    NOP;
    NOP;
    NOP;
    JUMP Wait_Here_For_IRQs;

/* ================================================
** CAN_TX_HANDLER
**
** ISR clears the interrupt request from MB8, writes
** new data to be sent, and requests to send again
**
** ================================================
*/

CAN_TX_HANDLER:
  [--SP] = (R7:6, P5:5);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P5.H = HI(CAN_MB_DATA2(8));
  P5.L = LO(CAN_MB_DATA2(8));
  R7 = W[P5](Z);  /* Retrieve Previously Sent Data */

  R6 = 0xFF;  /* Mask Upper Byte to Check Lower */
  R6 = R6 & R7;  /* Byte for Wrap */
  R5 = 0xFF;  /* Check Wrap Condition */

  CC = R6 == R5;  /* Check if Lower Byte Wraps */

  IF CC JUMP HANDLE_COUNT_WRAP;
```

```
    R7 += 1;  /* If no wrap, Increment Count */
    JUMP PREPARE_TO_SEND;

    HANDLE_COUNT_WRAP:
      R6 = 0xFF00(Z);  /* Mask Off Lower Byte */
      R7 = R7 & R6;  /* Sets Lower Byte to 0 */
      R6 = 0x0100(Z);  /* Increment Value for Upper Byte */
      R7 = R7 + R6;   /* Increment Upper Byte */

    PREPARE_TO_SEND:
      W[P5] = R7;  /* Set New TX Data */

      P5.L = LO(CAN_TRS1);
      R7 = TRS8;
      W[P5] = R7;  /* Issue New Transmit Request */

      P5.L = LO(CAN_MBTIF1);
      W[P5] = R7; /* Clear Interrupt Request Bit for MB08 */
      ASTAT = [SP++];  /* Restore Clobbered Registers */
      (R7:6, P5:5) = [SP++];
      SSYNC;
      RTI;

/* ================================================
** CAN_RX_HANDLER
**
** ISR clears the interrupt request from MB9, writes
** new data to be sent, and requests to send again
**
** ================================================
*/
```

```
CAN_RX_HANDLER:
  [--SP] = (R7:7, P5:4);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P4.H = CAN_RX_WORD;  /* Set Pointer to Storage Element */
  P4.L = CAN_RX_WORD;

  P5.H = HI(CAN_RMP1);
  P5.L = LO(CAN_RMP1);
  R7 = RMP9;
  W[P5] = R7;  /* Clear Message Pending Bit for MB09 */
  P5.L = LO(CAN_MBRIF1);
  W[P5] = R7; /* Clear Interrupt Request Bit for MB09 */

  P5.L = LO(CAN_MB_DATA3(9));
  R7 = W[P5](Z);  /* Read data from mailbox */
  W[P4] = R7;  /* Store data to SDRAM */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:7, P5:4) = [SP++];
  SSYNC;
  RTI;
```

20 TWO-WIRE INTERFACE CONTROLLERS

The two, Two-Wire Interface (TWI) controllers allow a device to interface to an Inter IC bus as specified by the *Philips I2C Bus Specification version 2.1* dated January 2000.

Overview

Each TWI is fully compatible with the widely used I^2C bus standard. They are designed with a high level of functionality and are compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth the TWI controllers can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

Each TWI externally moves 8-bit data while maintaining compliance with the I^2C bus protocol. The *Philips I2C Bus Specification version 2.1* covers many variants of I^2C. The TWI controllers include these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master 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 lock-up
- Input filter for spike suppression
- Serial Camera control Bus support as specified in *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*.

Table 20-1 shows the pins for the TWIs. Two bidirectional pins externally interface each TWI controller to the I^2C bus. The interface is simple and no other external connections or logic are required.

Table 20-1. TWI Pins

Pin	Description
SDAx	In/Out TWI serial data, high impedance reset value.
SCLx	In/Out TWI serial clock, high impedance reset value.

Architecture

Figure 20-1 illustrates the overall architecture of the TWI controllers.

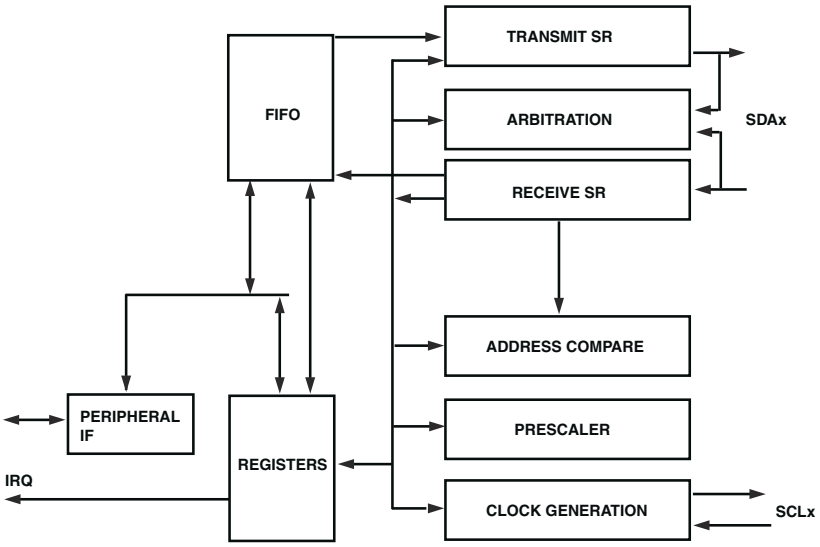


Figure 20-1. TWI Block Diagram

The peripheral interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer's model. Status bits can be updated by their respective functional blocks.

The FIFO buffer is configured as a1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event a TWI controller module is accessed as a slave.

The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for SCLx clock generation.

It is not always possible to achieve 10 MHz accuracy. In such cases, it is safe to round up the PRESCALE value to the next highest integer. For example, if SCLK is 133 MHz, the PRESCALE value is calculated as 133 MHz/10 MHz = 13.3. In this case, a PRESCALE value of 14 ensures that all timing requirements are met.

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.

Register Descriptions

Each TWI controller has 16 registers described in the following sections.

TWI Control (TWIx_CONTROL) Registers

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.

TWI Control Registers (TWIx_CONTROL)

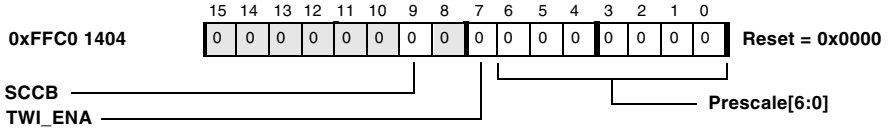


Figure 20-2. TWI Control Register

Additional information for the TWIx_CONTROL register bits are:

- **SCCB Compatibility** (SCCB).

  SCCB compatibility is an optional feature and should not be used in an I2C bus system. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

  [1] Master transfers are SCCB compatible. All slave asserted acknowledgement bits are ignored by this master.

  [0] Master transfers are not SCCB compatible.

- **TWI Enable** (TWI_ENA).

  This bit must be set for slave mode or master mode operation. It is recommended that this bit be set at the time PRESCALE is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

  [1] Internal time reference is enabled. Slave mode and master mode operation is reliable.

  [0] Internal time reference is disabled.

- **Prescale** (PRESCALE).

  The number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz.

TWI Clock Divider (TWIx_CLKDIV) Registers

During master mode operation, the SCLx clock divider register (TWIx_CLKDIV) values are used to create the high and low durations of the serial clock (SCLx). Serial clock frequencies can vary from 400 kHz to less than 20 kHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

CLKDIV = TWIx SCLx Period/10 MHz time reference

For example, for an SCLx of 400 kHz (period = 1/400 kHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

```
CLKDIV = 2500 ns / 100 ns = 25
```

For an SCLx with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8.

Note that CLKLOW and CLKHI add up to CLKDIV.

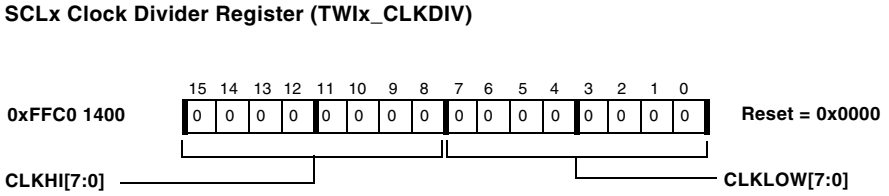


Figure 20-3. SCL Clock Divider Registers

Additional information for the TWIx_CLKDIV register bits includes:

- **Clock High** (CLKHI).

  Number of 10 MHz time reference periods the serial clock (SCLx) waits before a new clock low period begins, assuming a single master.

- **Clock Low** (CLKLO).

  Number of internal time reference periods the serial clock (SCLx) is held low.

TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers

The TWI slave mode control registers (TWIx_SLAVE_CTRL) control the logic associated with slave mode operation. Settings in these registers do not affect master mode operation and should not be modified to control master mode functionality.

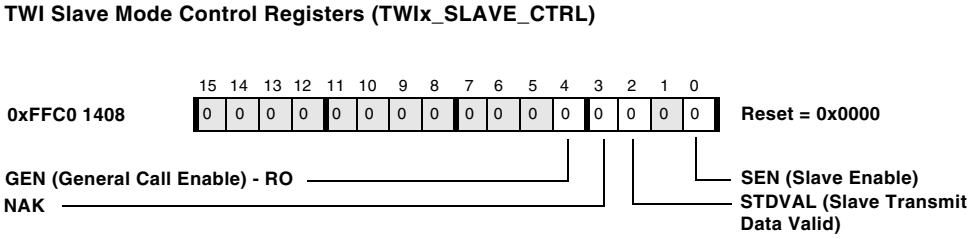


Figure 20-4. TWI Slave Mode Control Registers

Additional information for the TWIx_SLAVE_CTRL register bits includes:

- **General Call Enable** (GEN).

  General call address detection is available only when slave mode is enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

  [0] General call address matching is not enabled.

- **NAK** (NAK).

  [1] Slave receive transfers generate a data NAK (Not Acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

- Slave Transmit Data Valid (STDVAL).

  [1] Data in the transmit FIFO is available for a slave transmission.

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

- **Slave Enable** (SEN).

  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.

TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers

The TWI slave mode address registers (TWIx_SLAVE_ADDR) hold the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

TWI Slave Mode Address Registers (TWIx_SLAVE_ADDR)

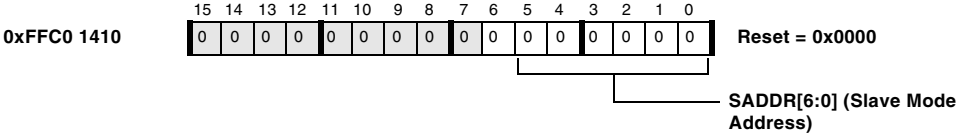


Figure 20-5. TWI Slave Mode Address Registers

TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers

During and at the conclusion of slave mode transfers, the TWI Slave Mode status registers (TWIx_SLAVE_STAT) hold information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

TWI Slave Mode Status Registers (TWIx_SLAVE_STAT)

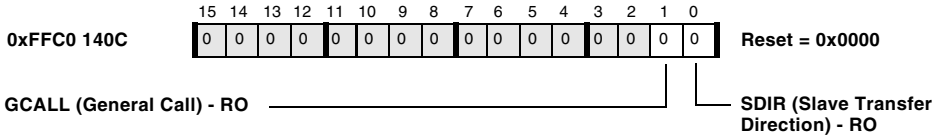


Figure 20-6. TWI Slave Mode Status Registers

Additional information for the TWIx_SLAVE_STAT register bits includes:

- **General Call** (GCALL).

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the address was determined to be a general call.

  [0] At the time of addressing, the address was not determined to be a general call.

- **Slave Transfer Direction** (SDIR).

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

TWI Master Mode Control (TWIx_MASTER_CTRL) Registers

The TWI Master Mode control registers (TWIx_MASTER_CTRL) control the logic associated with master mode operation. Bits in these registers do not affect slave mode operation and should not be modified to control slave mode functionality.

TWI Master Mode Control Registers (TWIx_MASTER_CTRL)

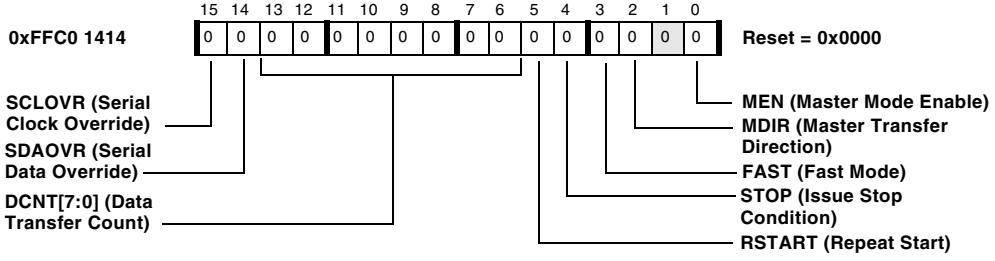


Figure 20-7. TWI Master Mode Control Registers

Additional information for the TWIx_MASTER_CTRL register bits includes:

- **Serial Clock Override** (SCLOVR).

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

  [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

  [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.

- **Serial Data (SDA) Override** (SDAOVR).

  This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

  [1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

  [0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

- **Data Transfer Count** (DCNT).

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat Start** (RSTART).

  [1] Issue a repeat Start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat Start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

  [0] Transfer concludes with a Stop condition.

- **Issue Stop Condition** (STOP).

  [1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt status registers (TWIx_INT_STAT) are updated.

  [0] Normal transfer operation.

- **Fast Mode** (FAST).

  [1] Fast mode (up to 400K bits/s) timing specifications in use.

  [0] Standard mode (up to 100K bits/s) timing specifications in use.

- **Master Transfer Direction** (MDIR).

  [1] The initiated transfer is master receive.

  [0] The initiated transfer is master transmit.

- **Master Mode Enable** (MEN).

  This bit self clears at the completion of a transfer (after the DCNT bit decrements to zero), including transfers terminated due to errors.

  [1] Master mode functionality is enabled. A Start condition is generated if the bus is idle.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDAx, SCLx) are no longer driven. Write-1-to-clear (W1C) status bits are not affected.

TWI Master Mode Address (TWIx_MASTER_ADDR) Registers

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI Master Mode Address registers (TWIx_MASTER_ADDR). When programming these registers, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to these registers. For example, if the slave address is b#1010000X, where X is the read/write bit, then TWIx_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the Master Mode control register.

TWI Master Mode Address Registers (TWIx_MASTER_ADDR)

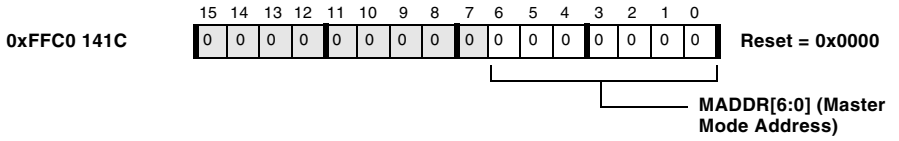


Figure 20-8. TWI Master Mode Address Registers

TWI Master Mode Status (TWIx_MASTER_STAT) Registers

The TWI Master Mode status registers (TWIx_MASTER_STAT) hold information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

TWI Master Mode Status Registers (TWIx_MASTER_STAT)

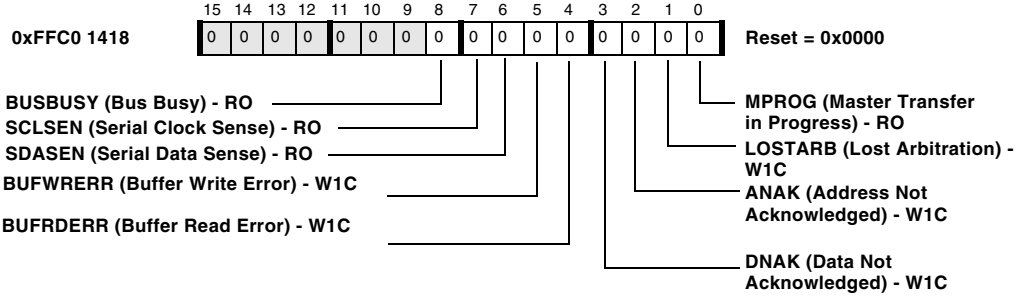


Figure 20-9. TWI Master Mode Status Registers

Additional information for the TWIx_MASTER_STAT registers bits includes:

- **Bus Busy** (BUSBUSY).

  Indicates whether the bus is currently busy or free. This indication is for all devices not this device alone. Upon a Start condition, the setting of the register value is delayed due to the input filtering. Upon a Stop condition the clearing of the register value occurs after t_{BUF}.

  [1] The bus is busy. Clock or data activity has been detected.

  [0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.

- **Serial Clock Sense** (SCLSEN).

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [1] An active “zero” is currently being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

  [0] An inactive “one” is currently being sensed on the serial clock.

- **Serial Data Sense** (SDASEN).

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

  [0] An inactive “one” is currently being sensed on the serial data line.

- **Buffer Write Error** (BUFWRERR).

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time.

  [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.

  [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.

  [0] The current master receive has not detected a NAK during data transmission.

- **Address Not Acknowledged** (ANAK).

  [1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer.

  [0] The current master transmit has not detected NAK during addressing.

- **Lost Arbitration** (LOSTARB).

  [1] The current transfer was aborted due to the loss of arbitration with another master.

  [0] The current transfer has not lost arbitration with another master.

- **Master Transfer in Progress** (MPROG).

  [1] A master transfer is in progress.

  [0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

TWI FIFO Control (TWIx_FIFO_CTRL) Registers

The TWI FIFO control register (TWIx_FIFO_CTRL) control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

TWI FIFO Control Registers (TWIx_FIFO_CTRL)

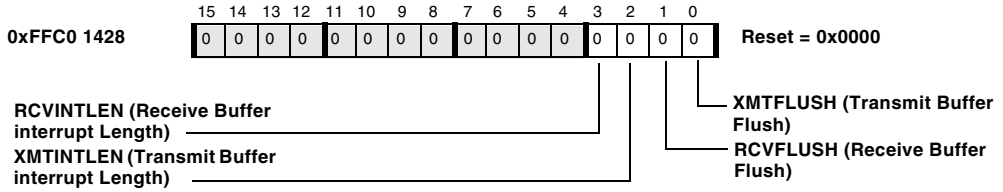


Figure 20-10. TWI FIFO Control Registers

Additional information for the TWIx_FIFO_CTRL 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 registers indicates two bytes in the FIFO are full (b#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 registers 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 to the as if the transmit buffer is empty.

  [0] Normal operation of the transmit buffer and its status bits.

TWI FIFO Status (TWIx_FIFO_STAT) Registers

The fields in the TWI FIFO status registers (TWIx_FIFO_STAT) indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

TWI FIFO Status Registers (TWIx_FIFO_STAT)
All bits are RO.

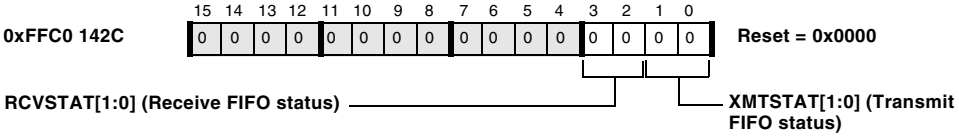


Figure 20-11. TWI FIFO Status Registers

Additional information for the TWIx_FIFO_STAT register bits includes:

- **Receive FIFO Status** (RCVSTAT).

  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).

  The XMTSTAT field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  b#11 – The FIFO is full and contains two bytes of data.

  b#10 – Reserved.

  b#01 – The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  b#00 – The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

TWI Interrupt Mask (TWIx_INT_ENABLE) Registers

The TWI interrupt mask registers (TWIx_INT_ENABLE) enable interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the TWI interrupt status (TWIx_INT_STAT) registers. Reading and writing the TWI interrupt mask registers does not affect the contents of the TWI interrupt status registers.

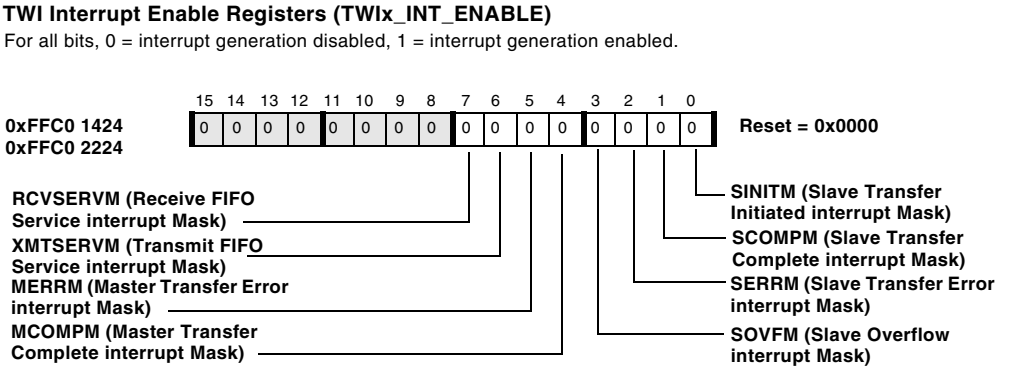


Figure 20-12. TWI Interrupt Mask Registers

Additional information for the TWIx_INT_ENABLE register bits includes:

- **Receive FIFO Service** (RCVSERV).

  If RCVINTLEN in the TWIx_FIFO_CTRL registers is 0, this bit is set each time the RCVSTAT field in the TWIx_FIFO_STAT registers is updated to either b#01 or b#11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to either b#10 or b#11.

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

- **Transmit FIFO Service** (XMTSERV).

  If XMTINTLEN in the TWIx_FIFO_CTRL registers is 0, this bit is set each time the XMTSTAT field in the TWIx_FIFO_STAT registers is updated to either b#01 or b#00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to b#00.

- **Master Transfer Error** (MERR).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of "one" in the corresponding interrupt source will result in asserting the interrupt output.

- **Master Transfer Complete** (MCOMP).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of "one" in the corresponding interrupt source will result in asserting the interrupt output.

- **Slave Overflow** (SOVF).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of "one" in the corresponding interrupt source will result in asserting the interrupt output.

- **Slave Transfer Error** (SERR).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of "one" in the corresponding interrupt source will result in asserting the interrupt output.

- **Slave Transfer Complete** (SCOMP).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

- **Slave Transfer Initiated** (SINIT).

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

TWI Interrupt Status (TWIx_INT_STAT) Registers

The TWI interrupt status registers (TWIx_INT_STAT) contain information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit.

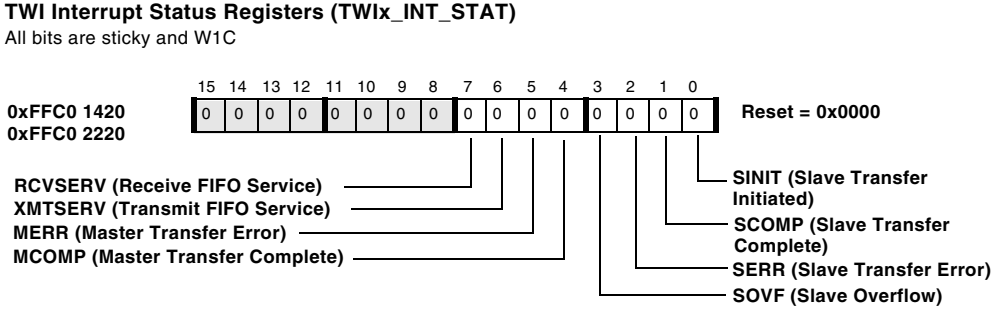


Figure 20-13. TWI Interrupt Status Registers

Additional information for the TWIx_INT_STAT register bits includes:

- **Receive FIFO Service** (RCVSERV).

  If RCVINTLEN in the TWIx_FIFO_CTRL registers is 0, this bit is set each time the RCVSTAT field in the TWIx_FIFO_STAT registers is updated to either b#01 or b#11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to either b#10 or b#11.

  [1] The receive FIFO buffer has one or two 8-bit locations containing data to be read.

  [0] FIFO does not require servicing or RCVSTAT field has not changed since this bit was last cleared.

- **Transmit FIFO Service** (XMTSERV).

  If XMTINTLEN in the TWIx_FIFO_CTRL registers is 0, this bit is set each time the XMTSTAT field in the TWIx_FIFO_STAT registers is updated to either b#01 or b#00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to b#00.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

  [0] FIFO does not require servicing or XMTSTAT field has not changed since this bit was last cleared.

- **Master Transfer Error** (MERR).

  [1] A master error has occurred. The conditions surrounding the error are indicated by the Master status registers (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 has been released.

  [0] The completion of a transfer has not been detected.

- **Slave Overflow** (SOVF).

  [1] The Slave Transfer Complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.

  [0] No overflow has been detected.

- **Slave Transfer Error** (SERR).

  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.

  [0] No errors have been detected.

- **Slave Transfer Complete** (SCOMP).

  [1] The transfer is complete and either a stop, or a restart was detected.

  [0] The completion of a transfer has not been detected.

- **Slave Transfer Initiated** (SINIT).

  [1] The slave has detected an address match and a transfer has been initiated.

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers

The TWI FIFO transmit data single byte registers (TWIx_XMT_DATA8) hold an 8-bit data value written into the FIFO buffer.

Transmit data is entered into the corresponding transmit buffer in a first-in first-out order. Although peripheral bus writes are 16 bits, a write access to TWIx_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

TWI FIFO Transmit Data Single Byte Registers (TWIx_XMT_DATA8)
All bits are WO. This register always reads as 0x0000.

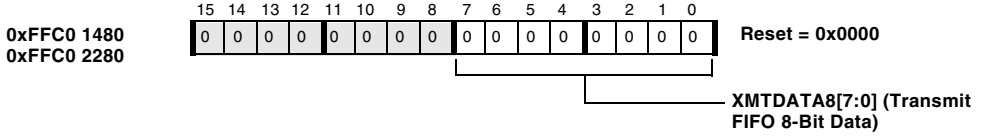


Figure 20-14. TWI FIFO Transmit Data Single Byte Registers

TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers

The TWI FIFO Transmit Data Double Byte registers (TWIx_XMT_DATA16) hold a 16-bit data value written into the FIFO buffer.

To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access.

The data is written in little endian byte order as shown in Figure 20-15 where Byte 0 is the first byte to be transferred and Byte 1 is the second byte to be transferred. With each access, the Transmit status (XMTSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

Figure 20-15. Little Endian Byte Order

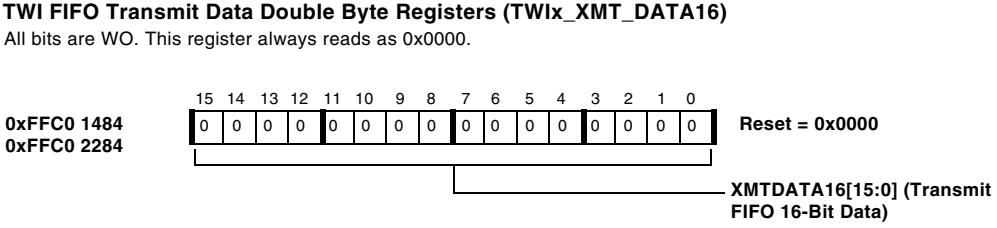


Figure 20-16. TWI FIFO Transmit Data Double Byte Registers

TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers

The TWI FIFO Receive Data Single Byte registers (TWIx_RCV_DATA8) hold an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWIx_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the Receive status (RCVSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.

TWI FIFO Receive Data Single Byte Registers (TWIx_RCV_DATA8)
All bits are RO.

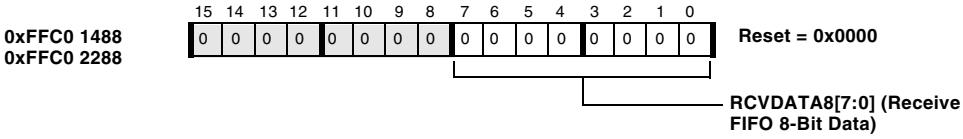


Figure 20-17. TWI FIFO Receive Data Single Byte Registers

TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers

The TWI FIFO Receive Data Double Byte registers (TWIx_RCV_DATA16) hold a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access. The data is read in little endian byte order as shown in Figure 20-18 where Byte 0 is the first byte received and Byte 1 is the second byte received. With each access, the Receive status (RCVSTAT) field in the TWIx_FIFO_STAT registers is updated to indicate it is empty. If an access is performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.

Figure 20-18. Little Endian Byte Order

TWI FIFO Receive Data Double Byte Registers (TWIx_RCV_DATA16)
All bits are WO.

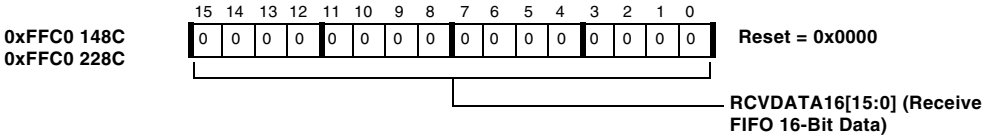


Figure 20-19. TWI FIFO Receive Data Double Byte Registers

Data Transfer Mechanics

The TWI controllers follow the transfer protocol of the *Philips I2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 20-20.

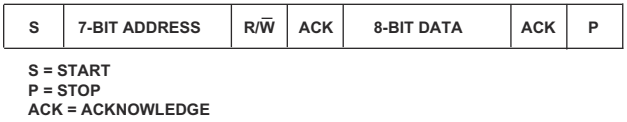


Figure 20-20. Basic Data Transfer

To better understand the mapping of the TWI controller register contents to a basic transfer, Figure 20-21 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI controller successfully transmits one byte of data. The slave has acknowledged both address and data.

Figure 20-21. Data Transfer With Bit Illustration

Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states. If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 20-22.

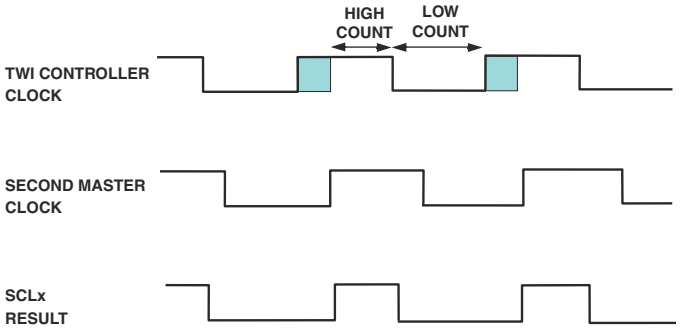


Figure 20-22. TWI Clock Synchronization

The TWI controller's serial clock (SCLx) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCLx line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controllers initiate a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 20-23.

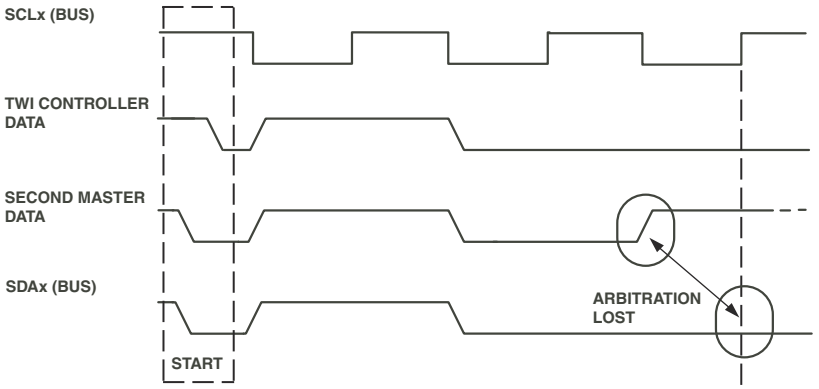


Figure 20-23. TWI Bus Arbitration

The TWI controllers monitor the serial data bus (SDAx) while SCLx is high and if SDA is determined to be an active logic 0 level while the TWI controller's data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time 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 20-24.

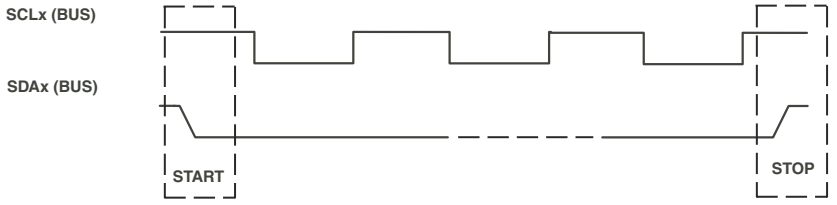


Figure 20-24. TWI Start and Stop Conditions

The TWI controller's special case Start and Stop conditions include:

- TWI controller addressed as a slave-receiver

  If the master asserts a Stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- TWI controller addressed as a slave-transmitter

  If the master asserts a Stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a Slave Transfer Error (SERR).

- TWI controller as a master-transmitter or master-receiver

  If the Stop bit is set during an active master transfer, the TWI controller issues a Stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

General Call Support

The TWI controllers always decode and acknowledge a general call address if it is enabled as a slave (SEN) and if General Call is enabled (GEN). General Call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

The byte following the general call address usually defines what action needs to be taken by the slaves in response to the call. The command in the second byte is interpreted based on the value of its LSB. For a TWI slave device, this is not applicable, and the bytes received after the general call address are considered data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most effected. When Fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a Stop and Start condition (t_{BUF})

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

General Setup

General Setup refers to register writes that are required for both slave mode operation and master mode operation. General Setup should be performed before either the master or slave enable bits are set.

Program the TWIx_CONTROL registers to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of f_{SCLK} / 10MHz

All values should be rounded up to the next whole number. The TWI_ENA bit enable must be set. Note once the TWI controller is enabled a bus busy condition may be detected. This condition should clear after t_{BUF} has expired assuming additional bus activity has not been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program 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_ENABLE. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.
4. Program TWIx_SLAVE_CTRL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 20-2 shows what the interaction between a TWI controller and the processor might look like using this example.

Table 20-2. Slave Mode Setup Interaction

TWI Controller Master	Processor
interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
interrupt: SCOMP – Slave transfer complete.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

Program `TWIx_CLKDIV`. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program `TWIx_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWIx_XMT_DATA8` or `TWIx_XMT_DATA16`. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.
3. Program `TWIx_FIFO_CTRL`. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8 bits) or with each 2 bytes transmitted (16 bits).

4. Program TWIx_INT_ENABLE. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

5. Program TWIx_MASTER_CTRL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 20-3 shows what the interaction between a TWI controller and the processor might look like using this example.

Table 20-3. Master Mode Transmit Setup Interaction

TWI Controller Master	Processor
interrupt: XMTEMPTY – Transmit buffer is empty.	Write transmit FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits.

Master Mode Receive

Follow these programming steps for a single master mode transmit:

1. Program TWIx_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.

2. Program TWIx_FIFO_CTRL. Indicate if receive FIFO buffer interrupts should occur with each byte received (8 bits) or with each 2 bytes received (16 bits).

3. Program TWIx_INT_ENABLE. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

4. Program TWIx_MASTER_CTRL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

After the TWI_DCNT bit is decremented to zero, the TWI master device sends a NAK to indicate to the slave transmitter that the bus should be released. This allows the master to send the STOP signal to terminate the transfer.

Table 20-4 shows what the interaction between a TWI controller and the processor might look like using this example.

Table 20-4. Master Mode Receive Setup Interaction

TWI Controller Master	Processor
interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
interrupt: MCOMP – Master transfer complete.	Read receive FIFO buffer. 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. The following sections contain information intended to be a guide to assist the programmer in their service routine development.

Transmit/Receive Repeated Start Sequence

Figure 20-25 illustrates a repeated start data transmit followed by a data receive sequence.

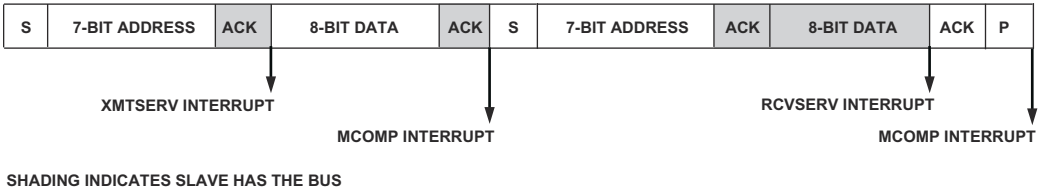


Figure 20-25. Transmit/Receive Data Repeated Start

The tasks performed at each interrupt are:

- `XMTSERV` interrupt

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, `FIFO_STATUS` would indicate the transmit FIFO is empty. When read, `DCNT` would be zero. Set the `RSTART` bit to indicate a repeated start and set the `MDIR` bit should a subsequent transfer be a data receive.

- `MCOMP` interrupt

  This interrupt was generated since all data has been transferred (`DCNT = 0`). If no errors were generated, a Start condition is initiated. Clear the `RSTART` bit and program the `DCNT` with the desired number of bytes to receive.

- `RCVSERV` interrupt

  This interrupt is generated due to the arrival of a byte into the receive FIFO. Simple data handling is all that is required.

- MCOMP interrupt

  The transfer is complete.

Receive/Transmit Repeated Start Sequence

Figure 20-26 illustrates a repeated start data receive followed by a data transmit sequence.

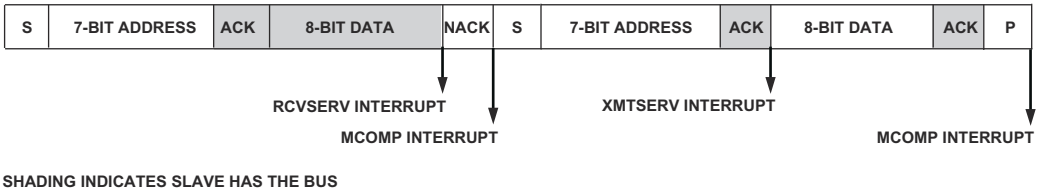


Figure 20-26. Receive/Transmit Data Repeated Start

The tasks performed at each interrupt are:

- RCVSERV interrupt

  This interrupt in generated due to the arrival of a data byte into the receive FIFO. Set the RSTART bit to indicate a repeated start and clear the MDIR bit should a subsequent transfer be a data transmit.

- MCOMP interrupt

  This interrupt has occurred due to the completion of the data receive transfer. If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to transmit.

- `XMTSERV` interrupt

  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- `MCOMP` interrupt

  The transfer is complete.

There is no timing constraint to meet the above conditions; the user can program the bits as required. Refer to "Clock Stretching During Repeated Start Condition" on page 20-48 for more on how the controller stretches the clock during repeated start transfers.

Clock Stretching

Clock stretching is an added functionality of the TWI controller in master mode operation. This new behavior utilizes self-induced stretching of the I^2C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this. The TWI Controller as master supports three modes of clock stretching:

- "Clock Stretching During FIFO Underflow" on page 20-45
- "Clock Stretching During FIFO Overflow" on page 20-46
- "Clock Stretching During Repeated Start Condition" on page 20-48

Clock Stretching During FIFO Underflow

During a master mode transmit, an interrupt is generated at the instant the transmit FIFO becomes empty. At this time, the most recent byte begins transmission. If the XMTSERV interrupt is not serviced, the concluding acknowledge phase of the transfer is stretched. Stretching of the clock continues until new data bytes are written to the transmit FIFO (TWI_XMT_DATA8 or TWI_XMT_DATA16). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (DCNT = 0) at which time the transmission is concluded (MCOMP) as shown in Figure 20-27 and described in Table 20-5.

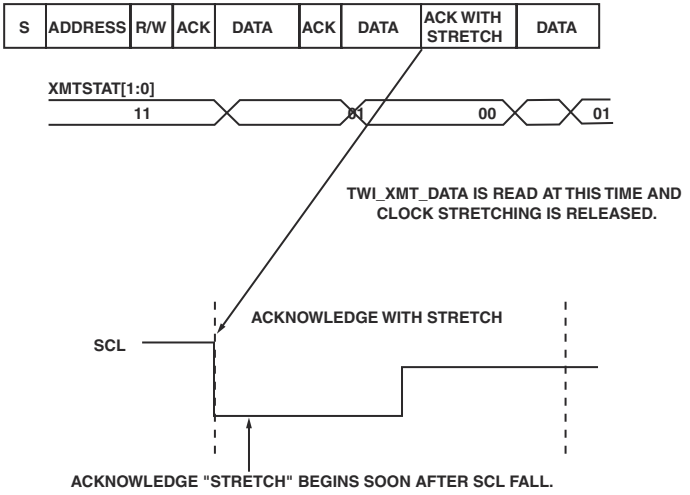


Figure 20-27. Clock Stretching During FIFO Underflow

Table 20-5. FIFO Underflow Case

TWI Controller	Processor
Interrupt: XMTSERV – Transmit FIFO buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).	Acknowledge: Clear interrupt source bits.

Clock Stretching During FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (`TWI_RCV_DATA8`, `TWI_RCV_DATA16`). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (`DCNT` = 0x00) at which time the reception is concluded (`MCOMP`) as shown in Figure 20-28 and described in Table 20-6.

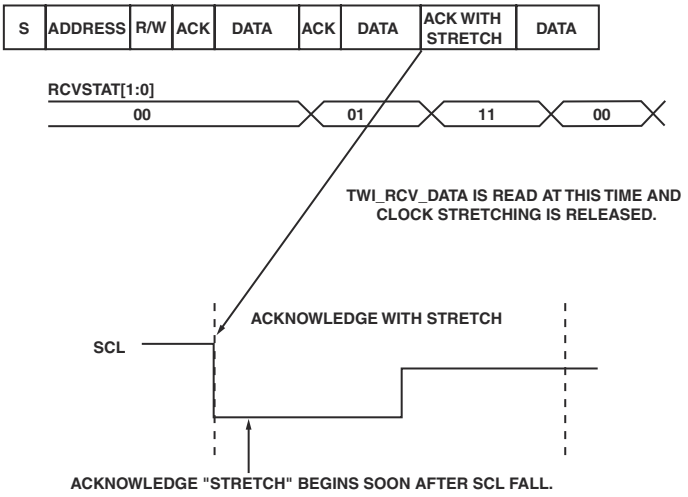


Figure 20-28. Clock Stretching During FIFO Overflow

Table 20-6. FIFO Overflow Case

TWI Controller	Processor
Interrupt: RCVSERV – Receive FIFO buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Clock Stretching During Repeated Start Condition

The repeated start feature in I^2C protocol requires transitioning between two subsequent transfers. With the use of clock stretching, the task of managing transitions becomes simpler, and common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock initiates a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (DCNT = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 20-29 and described in Table 20-7.

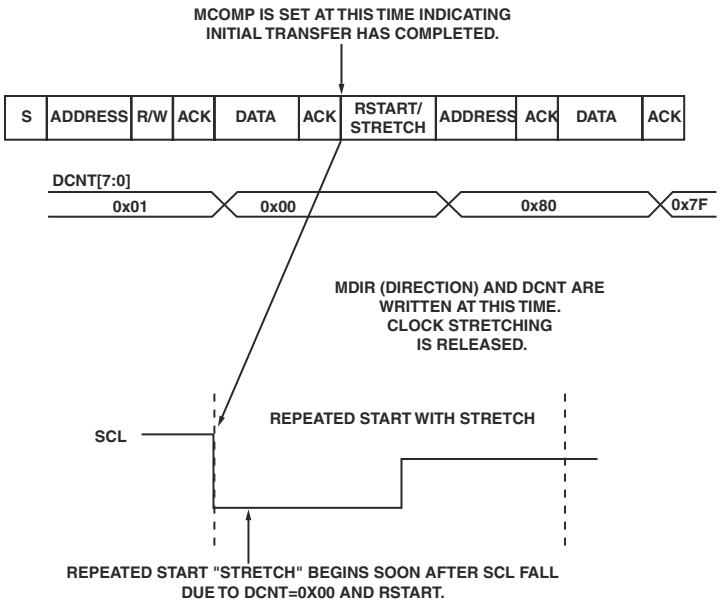


Figure 20-29. Clock Stretching During Repeated Start Condition

Table 20-7. 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 TWI_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).
Interrupt: RCVSERV – Receive FIFO is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I2C Bus Specification version 2.1* dated January 2000.

21 MEDIA TRANSCEIVER MODULE (MXVR)

The Media Transceiver module (MXVR) serves as the network interface to a Media Oriented System Transport (MOST) ring network for the ADSP-BF539 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.

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 (24 Mbps). 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-BF539 core fast and easy access to the data. Data is DMA'ed 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 fourteen dedicated DMA channels that work autonomously from the ADSP-BF539 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, two DMA channels support the transmission and reception of normal control messages, and two 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-BF539 nodes or other MOST transceivers.

Interface Signals

Table 21-1 on page 21-3 lists the MXVR signal pins. All output pins are 3.3V compliant. The MRX and $\overline{\text{MRXON}}$ input pins are 5V tolerant. All signal pins except for the dedicated crystal oscillator pins MXI and MXO, the $\overline{\text{MRXON}}$ input pin, and the analog MLF pin are multiplexed with GPIO signals. The selection of whether the pin has the MXVR functionality or GPIO functionality is set within the ADSP-BF539 GPIO Module. See Chapter 15, “General Purpose Input/Output Ports”

Table 21-2 on page 21-3 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.

The MXEVDD power pin supplies the MXVR crystal oscillator with 3.3V power. MXEVDD may be connected to the main V_{DDEXT} power plane. There should be local bypass capacitors connected between MXEVDD and MXEGND.

The MXEGND pin serves as the isolated ground for the MXVR PLL and MXVR crystal oscillator. MXEGND may be connected to the main ground plane. There should be local bypass capacitors connected both between the MXEVDD and MXEGND and between the MPIVDD and MXEGND.

Table 21-1. MXVR Signal Pins

Signal Name	Function	Direction	Alternate Function
MXI	MXVR Crystal Input	Input	
MXO	MXVR Crystal Output	Output	
MRX	MXVR Receive Data	Input (5V tolerant)	PC4[1]
MTX	MXVR Transmit Data	Output	PC5
MMCLK	MXVR Master Clock	Output	PC6
MBCLK	MXVR Bit Clock	Output	PC7
MFS	MXVR Frame Sync	Output	PC8
$\overline{\text{MTXON}}$	MXVR Transmit PHY On	Output	PC9
$\overline{\text{MRXON}}$	MXVR Receive PHY On	Input (5V tolerant)	
MLF	MXVR Loop Filter	Analog	

1 When configured as GPIO, PC4 behaves as an input/open-drain output.

The MPIVDD power pin supplies the MXVR PLL with 1.2V power. MPIVDD may be connected to the main V_{DDINT} power plane through a ferrite bead. There should be local bypass capacitors connected between MPIVDD and MXEGND.

Figure 21-1 on page 21-4 shows an example circuit connection of the ADSP-BF539 processor to a MOST network. This diagram is intended as an example, and exact connections and recommended circuit values should be obtained by contacting Analog Devices, Inc.

Table 21-2. MXVR Power Supply Pins

Signal Name	Function	Supply
MXEVDD	MXVR External Power Supply	3.3V Power
MXEGND	MXVR Ground (serves as ground for MXVR_PLL and crystal oscillator)	Ground
MPIVDD	MXVR Internal Power Supply	1.2V Power

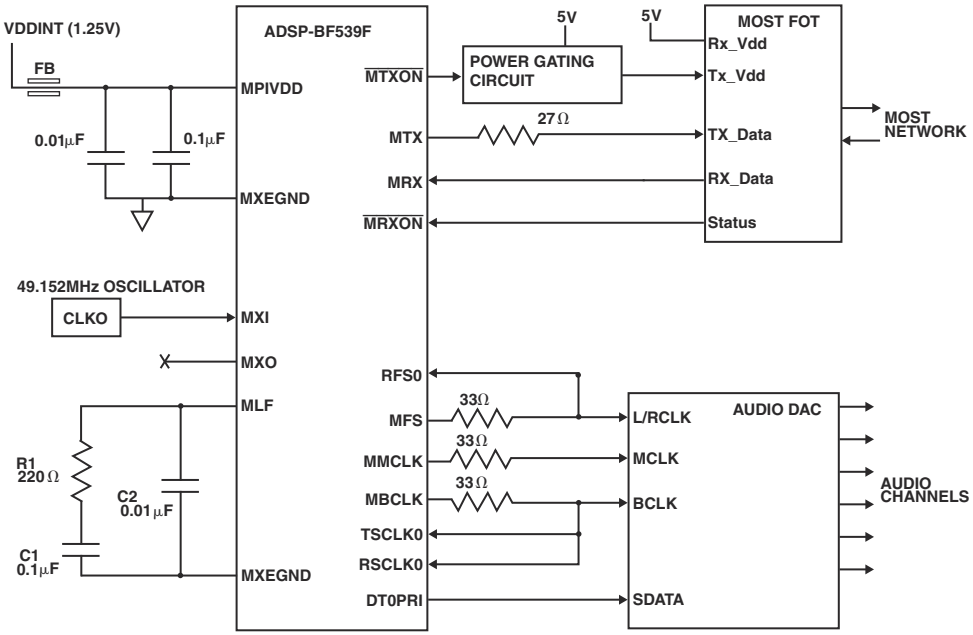


Figure 21-1. Example Connections of ADSP-BF539 to MOST Network

MXVR Memory Map

Table B-28 on page B-28 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 will generate a bus error exception and the data returned will be unknown. An incorrectly sized or misaligned write to an MMR will generate a hardware error interrupt and the write will not modify the MMR.

Register Descriptions

The following sections describe the registers of the MXVR module.

MXVR Configuration Register (MXVR_CONFIG)

The MXVR_CONFIG register sets the configuration of the MXVR node.

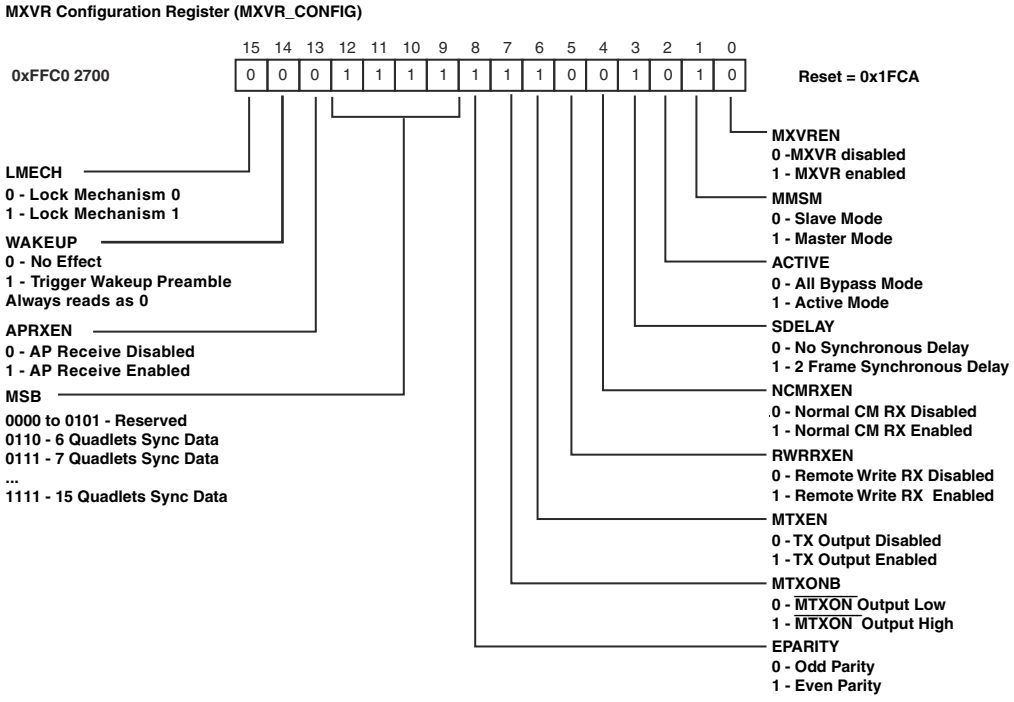


Figure 21-2. MXVR Configuration Register (MXVR_CONFIG)

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 will operate based on how the node has been configured in the MXVR MMR registers. Note that the MXVR should never be enabled without going through the PLL start-up sequence.

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 either a crystal at 1024 * Fs connected between the MXI and MXO pins or by an externally generated clock at 1024 * Fs driven onto the MXI pin. The transmit clock is then used to generate the data stream transmitted on the MTX pin. In addition, MXVR PLL 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 PLL 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 has been enabled. When in Active Mode the MXVR can transmit and receive data. When in All Bypass - MXVR Enabled Mode, the MRX input pin is directly connected to MTX and the MXVR can only receive data. When the MXVREN bit is set to "1" and the ACTIVE bit is set to "1", the MXVR will operate in Active Mode. When the MXVREN bit is set to "1" and the ACTIVE bit is set to "0", the MXVR will operate in All Bypass - MXVR Enabled Mode. When the MXVREN bit is set to "0", the ACTIVE bit has no meaning.

The Synchronous Data Delay (SDELAY) bit determines whether the synchronous data field will be delayed by two frames (SDELAY="1") or zero frames (SDELAY="0") passing through the MXVR. In the zero frame delay case the synchronous data will only be delayed by a few bit periods passing through the MXVR. If the MXVR is in All Bypass - MXVR Disabled Mode (MXVREN = "0") or in All Bypass - MXVR Enabled Mode (MXVREN = "1" and ACTIVE = "0"), the SDELAY bit has no meaning. If the MXVR is in Active Mode (MXVREN = "1" and ACTIVE = "1") and Master Mode (MMSM = "1"), there will always be two frame delays for synchronous data and the SDELAY bit will always read as "1". If the MXVR is in Active Mode (MXVREN = "1" and ACTIVE = "1"), Slave Mode (MMSM = "0"), and the SDELAY bit is set to "1", there will be two frame delays for synchronous data. If the MXVR is in Active Mode (MXVREN = "1" and ACTIVE = "1"), Slave Mode (MMSM = "0"), and the SDELAY bit is set to "0", there will only be a few bit delays for synchronous data. Note that synchronous data routing can only be done if the MXVR is the Master and is in Active Mode or if the MXVR is a Slave and is in Active Mode with the SDELAY bit is set to "1".

Table 21-3 on page 21-8 lists all possible operating modes of the MXVR, the bit encodings to select the modes, and the functionality in the modes.

Table 21-3. MXVR Operating Modes

Mode	MXVREN	MMSM	ACTIVE	SDELAY	Functionality
All Bypass - MXVR Disabled	0	X	X	X	-Bypassed from MRX to MTX -Cannot receive or transmit SD, AP, and CM -Cannot route or mute SD
Slave Node All Bypass - MXVR Enabled	1	0	0	X	-Bypassed from MRX to MTX -Can only receive SD, AP, and CM -Cannot route or mute SD
Slave Node Active Mode - Zero Frame Delay	1	0	1	0	-SD delayed by few bit periods -Can receive and transmit SD, AP, and CM -Cannot route SD, Can mute SD
Slave Node Active Mode - Two Frame Delay	1	0	1	1	-SD delayed by 2 frame periods -Can receive and transmit SD, AP, and CM -Can route and mute SD
Master Node Active Mode - Two Frame Delay	1	1	1	1	-SD delayed by 2 frame periods -Can receive and transmit SD, AP, and CM -Can route and mute SD
SD = Synchronous Data, AP = Asynchronous Packets, CM = Control Messages					

The Normal Control Message Receive Enable (NCMRXEN) bit determines whether the MXVR is enabled to receive Normal control messages. If the MXVR receives a Normal control message and the NCMRXEN bit is set to "1", the MXVR will write the received data into the Control Message Receive Buffer. If the MXVR receives a Normal control message when the NCMRXEN bit is set to "0", the MXVR will not respond to the Normal control message and will not write to the Control Message Receive Buffer. The NCMRXEN bit is reset to "0".

The Remote Write Receive Enable (RWRRXEN) bit determines whether the MXVR is enabled to receive Remote Write control messages. If the MXVR receives a Remote Write control message and the RWRRXEN bit is set to "1", the MXVR will write the received data into the Remote Read Buffer and will store the MAP and Length values. If the MXVR receives a

Remote Write control message when the RWRRXEN bit is set to "0", the MXVR will not write the received data to the Remote Read Buffer and will not write the MAP or Length value. In addition the MXVR will respond to the Remote Write control message with Transmission Status of 0x11 (Not Supported). The RWRRXEN bit is reset to "0".

The MXVR Transmit Data Enable (MTXEN) bit enables or disables the data stream transmitted on the MTX pin when the MXVR is enabled. If the MXVREN bit is set to "1" and the MTXEN bit is set to "0", the MTX pin will remain at a logic low level. If the MXVREN bit is set to "1" and the MTXEN bit is set to "1", the MTX pin will output the transmitted data stream. If the MXVREN bit is set to "0", the MXVR will be in All Bypass - MXVR Disabled Mode and the MTXEN bit has no meaning.

The MXVR PHY Transmitter On (MTXONB) bit sets the state of the $\overline{\text{MTXON}}$ output pin. The $\overline{\text{MTXON}}$ output pin can be connected to the base of an external PNP bipolar transistor to gate on and off the power supplied to the PHY Transmitter (in the case of MOST, the Transmit FOT). If the MTXONB bit is set to "0", the $\overline{\text{MTXON}}$ output pin will be at a logic-low level (powering the PHY Transmitter on). If the MTXONB bit is set to "1", the $\overline{\text{MTXON}}$ output pin will be at a logic-high level (powering the PHY Transmitter off). The MTXONB bit has no other effect on the MXVR. The MTXONB bit is reset to “1”, so that the PHY Transmitter is off after the ADSP-BF539 processor has been reset.

The MXVR Even Parity Select (EPARITY) bit indicates whether the parity bit in the frame should be generated with Even Parity or Odd Parity. If the EPARITY bit is set to "0", Odd Parity will be selected. If the EPARITY bit is set to "1", Even Parity will be selected. For MOST, Even Parity should always be selected. The EPARITY bit is reset to "1".

The synchronous boundary value transmitted by the Master node in a network determines how many quadlets in the frame are dedicated to synchronous data and how many quadlets are dedicated to asynchronous packet data. A quadlet is 4 bytes of data or 4 physical channels in the frame. There are a total of 15 quadlets for synchronous and asynchronous

data in the frame. If the synchronous boundary is 6, then 24 bytes will be dedicated to synchronous data and 36 bytes will be dedicated to asynchronous packet data. The MXVR is capable of operating with a synchronous boundary from 0 to 15; however, the MOST specification limits the synchronous boundary to the range 6 to 14.

When the MXVR is in Master Mode, the value written to the MSB field will be transmitted over the network to all of the slaves as the synchronous boundary. When the MXVR is in Slave Mode, the MSB field is not used. When the MXVR is in either Master Mode or Slave Mode, the synchronous boundary value which was received by the node over the network can be observed as the RSB field in MXVR_STATE_0 register.

The Synchronous Boundary (MSB) field is writable if the MXVR is in Master Mode (MMSM = "1") and is read-only if the MXVR is in Slave Mode (MMSM = "0"). Writes to the MSB while in Slave Mode will be ignored and the MSB value will not be effected. Note that a particular procedure must be followed to dynamically change the synchronous boundary for the network to ensure that no data is corrupted and the asynchronous packet channel does not hang.

The Asynchronous Packet Receive Enable (APRXEN) bit determines whether the MXVR is enabled to receive Asynchronous Packets. If the MXVR receives an Asynchronous Packet and the APRXEN bit is set to "1", the MXVR will write the received data into the Asynchronous Packet Receive Buffer. If the MXVR receives an Asynchronous Packet when the APRXEN bit is set to "0", the MXVR will not write the packet to the Asynchronous Packet Receive Buffer. The APRXEN bit is reset to "0".

The Wake-Up (WAKEUP) bit is used to trigger the MXVR when in Master Mode to send the wake-up preamble which will indicate to any node in low-power mode to wake-up. If the MMSM bit is set to "1", writing a “1” to the WAKEUP bit will trigger the MXVR to send the wake-up preamble on the network. If MMSM is set to "0", writing a "1" to the WAKEUP bit will have no effect. Writing a "0" to the WAKEUP bit will have no effect. The WAKEUP bit will always read as "0".

The Lock Mechanism Select (LMECH) bit determines in what order the MXVR Master will send network preambles while locking the network. Lock Mechanism 0 provides the fastest lock time from the completely unlocked state to the super block locked state in a network with only MXVR nodes. Lock Mechanism 1 takes longer than Lock Mechanism 0 to go from the completely unlocked state to the super block locked state; however, if a node in the ring causes an unlock (i.e. 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 PLL Control Register 0 (MXVR_PLL_CTL_0)

The MXVR_PLL_CTL_0 register controls the crystal oscillator, the clock recovery PLL, and the clock outputs.

The MXVR Crystal Oscillator Clock Enable (MXTALCEN) bit enables or disables the clock output from the MXVR Crystal Oscillator which is used by the MXVR PLL. 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 PLL.

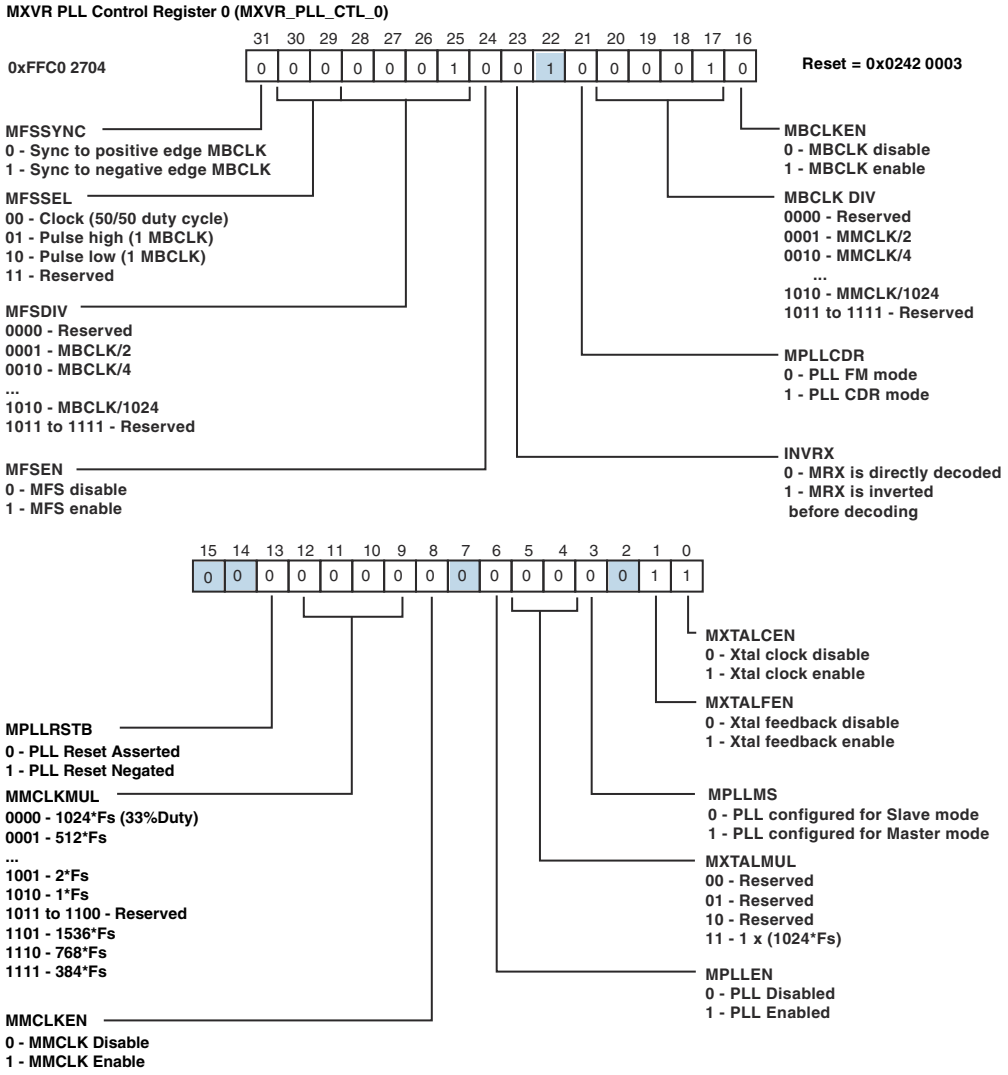


Figure 21-3. MXVR PLL Control Register 0 (MXVR_PLL_CTL_0)

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 PLL. 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 PLL Master/Slave Configuration (MPLLMS) bit configures the MXVR PLL connections for Master Mode or Slave Mode operation. At the beginning of the PLL start-up sequence the MPLLMS bit must be set to "1" if the MXVR is to operate in Master Mode or must be set to "0" if the MXVR is to operate in Slave Mode. The MPLLMS bit should never be changed while the MXVR PLL is enabled. The MPLLMS bit is reset to "0".

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 must be 1024 * Fs. If a crystal is placed between MXI and MXO, and the network will be disabled for an extended period of time, the MXTALFEN and the MXTALCEN can be set to "0" to decrease power consumption. If a clock is being directly driven to the MXI pin, and the network will be disabled for an extended period of time, the MXTALCEN can be set to "0" to decrease power consumption. However, the clock or crystal must be stable at frequency prior to starting up the MXVR PLL in order to lock the network.

The Crystal Multiplier (MXTALMUL) field determines the multiplication factor that will be used when the MXVR PLL is configured to multiply the crystal or input clock frequency up to the network clock frequency (1024 * Fs). Table 21-4 shows all the crystal frequencies for the sample frequencies of 38 kHz, 44.1 kHz, and 48 kHz. Note that the reset value for the MXTALMUL field is b#00 and the MXTALMUL field must be programmed to b#11 to select 1024 * Fs for proper operation.

Table 21-4. Crystal Input Frequencies

MXTALMUL	Crystal Frequency	Crystal Frequency Needed for Desired Fs		
		Fs = 38 kHz	Fs = 44.1 kHz	Fs = 48 kHz
b#00, b#01, b#10	Reserved	Reserved	Reserved	Reserved
b#11	1024 * Fs	38.912 MHz	45.1584 MHz	49.152 MHz

The MXVR PLL Enable (MPLLEN) bit enables or disables MXVR PLL. If the MPLLEN bit is set to a "0", the MXVR PLL is disabled and the PLL does not draw power. If the MPLLEN bit is set to a "1", the MXVR PLL is enabled and powered up. The MXVR PLL can be enabled or disabled by the MPLLEN bit independent of the state of the MXVREN bit. The PLL can be disabled in order to save power. The Start-Up Sequence for the MXVR PLL is described in “PLL Start-Up Sequence” on page 21-108. The MPLLEN bit is set to "0" by reset.

The MXVR PLL Reset (MPLLRSTB) bit resets MXVR PLL. When the MPLLRSTB bit is set to "0" the MXVR PLL is held in reset. When the MPLLRSTB bit is set to "1" the MXVR PLL is released from reset. The Start-Up Sequence for the MXVR PLL is described in “PLL Start-Up Sequence” on page 21-108. The MPLLRSTB bit is set to "0" by reset.

The MXVR PLL Clock/Data Recovery Mode (MPLLCDR) bit determines whether MXVR PLL is in Clock/Data Recovery Mode or in Frequency Multiplication Mode. When the MPLLCDR bit is set to "1" the MXVR PLL will recover the clock and data from the MRX input pin. When the MPLLCDR bit is set to "0" the MXVR PLL will multiply the frequency of the clock

supplied on the MXI pin. The Start-Up Sequence for the MXVR PLL is described in “PLL Start-Up Sequence” on page 21-108. The MPLLCDR bit is reset to "0".

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. Note that when reset is asserted, the MMCLK output pin will be three-stated. After reset is negated the MMCLK output pin will be driven low and will not toggle until the MXVR PLL has been started-up and the MMCLKEN bit has been set to “1”.

The Master Clock Multiplication Factor (MMCLKMUL) field determines the frequency of the MXVR Master Clock output pin (MMCLK). When the MXVR PLL is locked, the clock output on the MMCLK pin is generated by the clock recovered from the network data stream. The MMCLK clock frequency can be specified as a multiplication of the sample rate (Fs). The frequency can be set to be n * Fs where n can be 1, 2, 4, 8, 16, 32, 64, 128, 256, 384, 512, 768, 1024, and 1536. When MMCLKMUL is set to any value except 1024 * Fs or 1536 * Fs, the MMCLK duty cycle will be 50%. When MMCLKMUL is set to 1024 * Fs or 1536 * Fs, the MMCLK duty cycle will be 33%. Note that the frequency of MMCLK should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to "0".

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. Note that when reset is asserted, the MBCLK output pin will be three-stated. After reset is negated, the MBCLK output pin will be driven low and will not toggle until the MXVR PLL has been started-up and the MBCLKEN bit has been set to “1”.

The Bit Clock Divide Factor (MBCLKDIV) field determines the frequency of the MXVR Bit Clock output pin (MBCLK). The clock output on the MBCLK pin is generated by the MXVR Master Clock. The MBCLK clock frequency can be specified as a division of the MXVR Master Clock frequency. The frequency can be set to be f_{MMCLK} / n where n can be 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024. When MMCLKMUL is set to any value except 1024 * Fs or 1536 * Fs, the rising edge of MBCLK occurs in sync with the rising edge of MMCLK. When MMCLKMUL is set to 1024 * Fs or 1536 * Fs, the rising edge of MBCLK occurs in sync with the falling edge of MMCLK. Note that he 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 input pin (MRX) will be decoded as is or whether the data stream will be inverted before being decoded. If the INVRX bit is set to "0", the data stream will be decoded as is. If the INVRX bit is set to a "1", the data stream will be inverted prior to being decoded.

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. Note that when reset is asserted, the MFS output pin will be three-stated. After reset is negated the MFS output pin will be driven to it's inactive state and will not toggle until the MXVR PLL has been started-up and the MFSEN bit has been set to "1".

The Frame Sync Divide Factor (MFSDIV) field determines the frequency of the MXVR Frame Sync output pin (MFS). The clock output on the MFS pin is generated by the MXVR Bit Clock. The MFS clock frequency can be specified as a division of the MXVR Bit Clock frequency. The frequency can be set to be f_{MBCLK} / n where n can be 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024. Note that the MFSDIV should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to "0".

The Frame Sync Select (MFSSEL) field determines whether the MXVR Frame Sync output pin (MFS) will generate a 50% duty cycle clock, an active-high pulse, or an active-low pulse. If the MFSSEL field is set to b#00, the MFS will generate a 50% duty cycle clock. If the MFSSEL field is set to b#01, the MFS will generate an active-high pulse with a pulse length equal to the MBCLK period. If the MFSSEL field is set to b#10, the MFS will generate an active-low pulse with a pulse length equal to the MBCLK period. Note that the MFSSEL should only be changed when MMCLKEN, MBCLKEN, and MFSEN are all set to "0".

The Frame Sync Synchronization Select (MFSSYNC) bit determines the synchronization between the MFS and MBCLK output pins. If the MFS is programmed to be a 50% duty cycle clock (MFSSEL = b#00) or an active-high pulse (MFSSEL = b#01) and the MFSSYNC is set to "0", the rising edge of MFS occurs in sync with the falling edge of the MBCLK. If the MFS is programmed to be a 50% duty cycle clock (MCLKSEL = b#00) or an active-high pulse (MCLKSEL = b#01) and the MFSSYNC is set to "1", the rising edge of MFS occurs in sync with the rising edge of the MBCLK. If MFS is programmed to be a active-low pulse (MFSSEL = b#10) and the MFSSYNC is set to "0", the falling edge of MFS occurs in sync with the falling edge of MBCLK. If MFS is programmed to be a active-low pulse (MFSSEL = b#10) and the MFSSYNC is set to "1", the falling edge of MFS occurs in sync with the rising edge of MBCLK.

MXVR PLL Control Register 1 (MXVR_PLL_CTL_1)

The MXVR_PLL_CTL_1 register controls the PLL counter and some functions of the MXVR PLL.

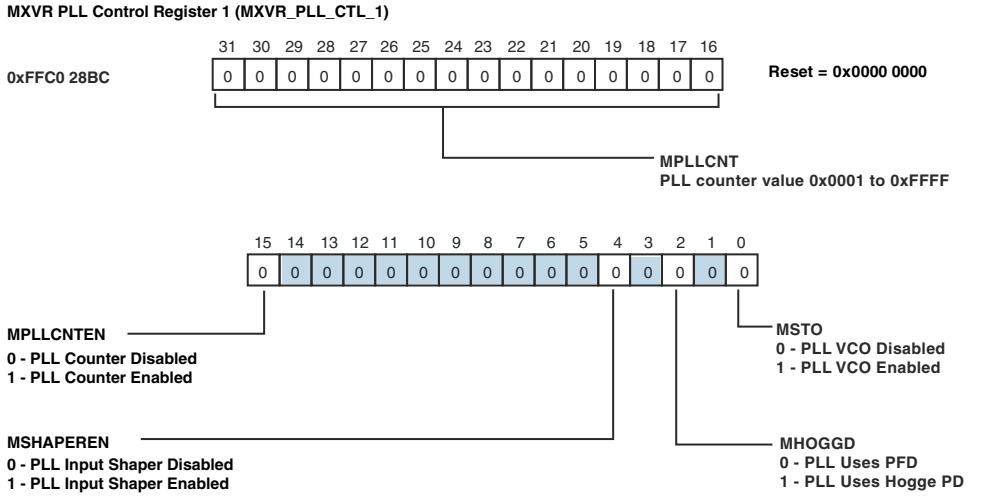


Figure 21-4. MXVR PLL Control Register 1 (MXVR_PLL_CTL_1)

The MXVR PLL Start Count Timeout (MSTO) bit enables or disables the VCO in MXVR PLL. When MXVR PLL is first enabled and released from reset, the loop filter must be given time to charge up prior to enabling the VCO. When MSTO is set to “0”, the VCO in MXVR PLL is disabled and when MSTO is set to “1”, the VCO in MXVR PLL is enabled. The MSTO bit is set to “0” by reset.

The MXVR PLL Hogge Phase Detector Select (MHOGGD) bit selects either the Hogge Phase Detector or the Phase-Frequency Detector for MXVR PLL. When the MHOGGD is set to “0”, MXVR PLL uses the Phase-Frequency Detector, and when MHOGGD is set to “1”, MXVR PLL uses the Hogge Phase Detector. The MHOGGD bit is set to “0” by reset.

The MXVR PLL Input Shaper Enable (MSHAPEREN) bit enables or disables the Input Shaper circuit for MXVR PLL. When MSHAPEREN bit is set to “0”, the Input Shaper is disabled in MXVR PLL. When MSHAPEREN bit is set to “1”, the Input Shaper is enabled in MXVR PLL. The MSHAPEREN bit is set to “0” by reset.

The MXVR PLL Counter Enable (MPLLCNTEN) bit enables or disables the PLL Counter. The PLL Counter is a 16-bit timeout counter clocked by the System Clock (SCLK) and generates the PCZ interrupt event when the PLL Counter decrements to zero. The PLL Counter is meant to be used as a timer for the MXVR PLL start-up sequence (hence the name "PLL Counter"). Since the PLL Counter operates based on SCLK, the PLL Counter works independently of the state of the MXVR (enabled, disabled, frame locked or unlocked) and of the state of the MXVR PLL (enabled, disabled, at frequency, not at frequency). The MPLLCNTEN bit must be set to "0" prior to starting the PLL Counter in order to reset the counter. When the MXVR_PLL_CTL_1 register is written setting the MPLLCNTEN bit to "1" and setting the MPLLCNT value, the PLL Counter will start decrementing. Once the PLL Counter reaches zero the PCZ interrupt event will change to "1" and can conditionally generate a Status interrupt. If the MPLLCNTEN bit is set to "0" after the PLL Counter has started decrementing but prior to the PLL Counter reaching zero, the PLL Counter will stop and no PCZ interrupt event will occur.

The MXVR PLL Count Value (MPLLCNT) is a 16-bit field which determines the timeout period of the PLL Counter. The PLL Counter has a prescaler which effectively multiplies the MPLLCNT value by 64 to determine the number of SCLK periods to count. The timeout period is determined by the following equation:

PLL Counter Timeout Period = $1/f_{SCLK}$ * 64 * MPLLCNT

The MPLLCNT value will be loaded into the PLL Counter when the MPLLCNTEN bit is set to "1". The desired MPLLCNT value must be written in to the MXVR_PLL_CTL_1 register in the same write as when the MPLLCNTEN

bit is set to "1". Once enabled, reading the MPLLCNT field will return the current state of the PLL Counter. The MPLLCNTEN bit must always be set to "0" before restarting the PLL Counter.

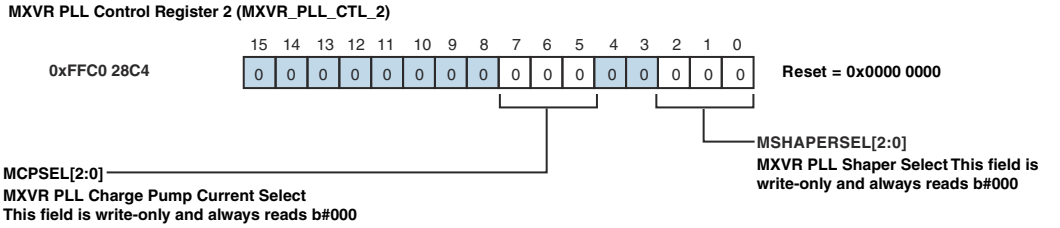


Figure 21-5. MXVR PLL Control Register 2 (MXVR_PLL_CTL_2)

MXVR PLL Control Register 2 (MXVR_PLL_CTL_2)

The MXVR_PLL_CTL_2 register allows control of the settings for the MXVR PLL Charge Pump Current and the MXVR PLL Shaper.

The MXVR PLL Charge Pump Current Select (MCPSEL) field controls the charge pump current in the MXVR PLL. This field is write-only and always reads as b#000.

The MXVR PLL Shaper Select (MSHAPERSEL) field controls the amount of pulse width distortion correction to be made to the incoming data stream when the MSHAPEREN bit is set to 1 in the MXVR_PLL_CTL_2 register. This field is write-only and always reads as b#000.

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.

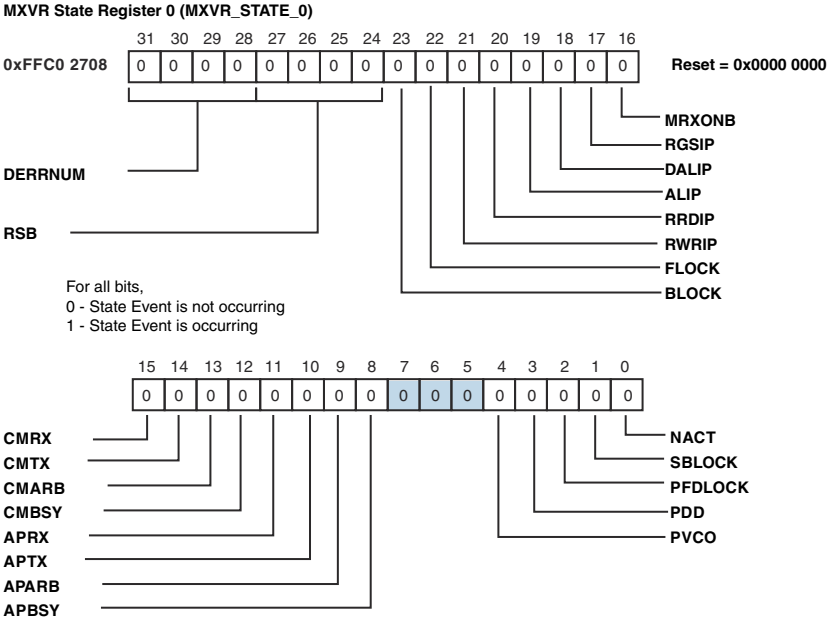


Figure 21-6. MXVR State Register (MXVR_STATE_0)

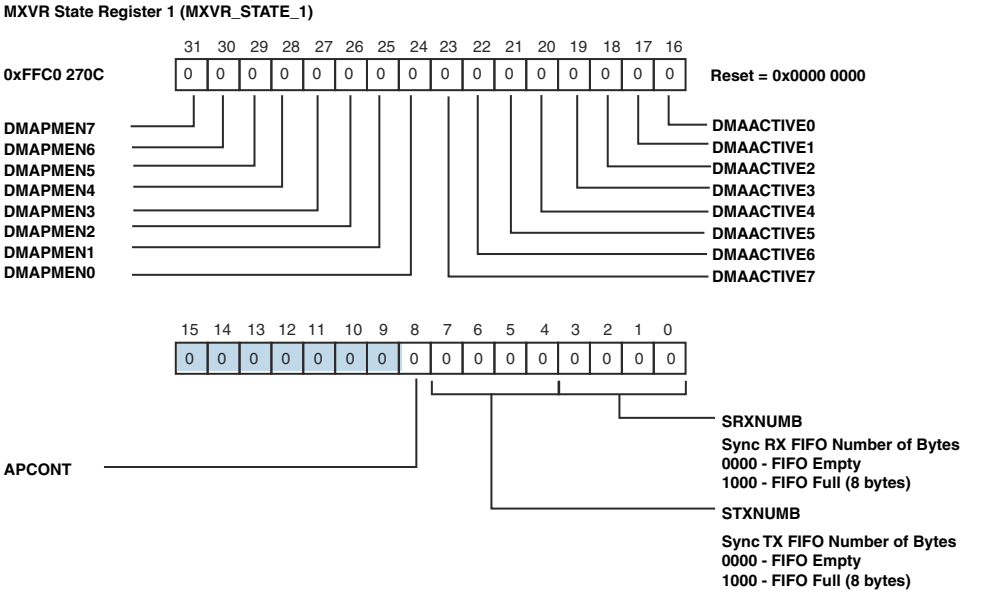


Figure 21-7. MXVR State Register (MXVR_STATE_1)

The MXVR PLL Charge Pump Current Select (CPSEL) field controls the charge pump current in the MXVR PLL. This field is write-only and always reads as b#000.

The MXVR PLL Shaper Select (MSHAPERSEL) field controls the amount of pulse width distortion correction to be made to the incoming data stream when the MSHAPEREN bit is set to 1 in the MXVR_PLL_CTL_2 register. This field is write-only and always reads as b#000.

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 Clock Recovery PLL has been 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 datastream, 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 PLL Phase Frequency Detector Lock (PFDLOCK) bit is a read-only bit which gives the state of the output of the MXVR PLL Phase Frequency Detector lock detect circuit. When the circuit detects a locked state the PFDLOCK bit changes to "1". When the circuit detects an unlocked state the PFDLOCK bit changes to "0". Note that this bit is for informational purposes only. The FLOCK, BLOCK, and SBLOCK bits are the recommended method for determining network lock status.

The MXVR PLL Data Detect (PDD) bit is a read-only bit which gives the state of the output of the MXVR PLL Data Detector circuit. When the circuit detects data being received the PDD bit changes to "1". When the circuit detects no data being received the PDD bit changes to "0". Note that this bit is for informational purposes only. The FLOCK, BLOCK, and SBLOCK bits are the recommended method for determining network lock status.

The MXVR PLL VCO On (PVCO) bit is a read-only bit which indicates whether the MXVR PLL VCO has started up. When the VCO is running, the PVCO bit will be "1". When the VCO is not running, the PVCO bit will be "0". Note that this bit is for informational purposes only. The FLOCK, BLOCK, and SBLOCK bits are the recommended method for determining network lock status.

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 has been 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 has been 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 has been 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 has been transmitted or once the Control Message being transmitted is successfully cancelled. Note that the Control Message Transmit Buffer should never be modified when the CMBSY bit is a “1”.

The Control Message Arbitrating (CMARB) bit is a read-only bit that indicates when the MXVR is arbitrating for the control message channel so that a control message can be transmitted. If the CMARB bit is a "1", the MXVR is arbitrating for the control message channel. If the CMARB bit is a "0", the MXVR is not arbitrating for the control message channel. While the MXVR is arbitrating for the control message channel, the current control message transmission can be cancelled. However, once the MXVR has won arbitration and the control message is being transmitted, the transmission cannot be cancelled. Note that when an attempt is made to cancel a control message, due to delays in clock synchronization and delays in reading and writing MMRs, the CMTS and CMTC bits in the MXVR_INT_STAT_1 register should be used to verify whether or not the control message was successfully cancelled.

The Control Message Transmitting (CMTX) bit is a read-only bit that indicates when the MXVR is actively transmitting a control message. If the CMTX bit is a "1", the MXVR has won arbitration and is in the process of transmitting a control message. If the CMTX bit is a "0", the MXVR is not in the process of transmitting a control message. Once the MXVR has started transmitting a control message, the current control message transmission cannot be cancelled.

The Receiving Control Message (CMRX) bit is a read-only bit that indicates when the MXVR is actively receiving a Normal control message. If the CMRX bit is a "1", the MXVR is in the process of receiving a Normal control message. If the CMRX bit is a "0", the MXVR is not in the process of receiving a Normal control message. Note that the Control Message Received (CMR) bit in the MXVR_INT_STAT_0 register will change to "1" when the reception of a Normal control message has successfully completed and can optionally generate an interrupt.

The $\overline{\text{MRXON}}$ Input Pin State (MRXONB) bit is a read-only bit which gives the current state of the $\overline{\text{MRXON}}$ input pin. The $\overline{\text{MRXON}}$ input pin should be connected to the optical PHY status output which indicates whether the PHY is currently receiving data. If the PHY is receiving data, the $\overline{\text{MRXON}}$ input pin should be driven to "0", If the PHY is not receiving data, the $\overline{\text{MRXON}}$ input pin should 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-BF539 processor 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 PLL 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 PLL has locked onto the received data, the MXVR has Frame Locked, and the MXVR has received two block preambles in the correct position. When the BLOCK bit is a “1”, the MXVR is Block Locked and when the BLOCK bit is a “0”, the MXVR is not Block Locked. Once an MXVR node is Block Locked and the ring is closed, control messages can be reliably transmitted and received by all nodes in the ring. Note that the MXVR can be Block Locked even if the ring network is not closed.

The Receive Synchronous Boundary (RSB) field is a read-only field which gives the synchronous boundary value received in the incoming datastream by the MXVR. The RSB value is only valid when the MXVR is Frame Locked.

The DMA Error Channel Number (DERRNUM) field is a read-only field which indicates which DMA Channel caused the last DMA Error (DERR) interrupt event. Table 21-5 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 21-5. 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 DMAACTIVEx bits indicate whether the DMA channel is active or inactive. When the DMAACTIVEx bit is “1”, DMA channel x is active and when the DMAACTIVEx bit is “0”, DMA channel x is inactive. Once the MDMAENx bit has been set to “1”, the exact time when the DMA goes active depends on the Flow Mode selected. When the MDMAENx bit is set to “1” in Stop Mode, the DMA channel will go active on the next frame boundary reached and will stop when the number of programmed transfers is complete. When the MDMAENx bit is set to “1” in Autobuffer Mode, the DMA channel will go active on the next frame boundary and will continue indefinitely. When the MDMAENx bit is set to “1” in Packet-Fixed Count Mode, Packet-Variable Count Mode, or Packet-Start/Stop Mode, the DMA will go active once DMAPMENx is “1” and the "start pattern" is found. When the DMA channel is active in Packet-Fixed Count Mode, the DMA channel will go inactive when the programmed number of transfers is done, When the DMA channel is active in Packet-Fixed Count Mode, the DMA channel will go inactive when the number of transfers specified in the packet are done. When the DMA channel is active in Packet-Start/Stop Mode, the DMA channel will go inactive when the "stop pattern" is found (Packet-Start/Stop). In any flow mode if the MDMAENx bit is set to “0”, the DMA channel will go inactive and disable on the next frame boundary reached.

The DMAPMENx bits indicate whether the DMA channel is enabled for Pattern Matching. In Packet-Fixed Count Mode, Packet-Variable Count Mode, or Packet-Start/Stop Mode, when the MDMAENx bit is set to “1”, the DMA channel will be enabled for pattern matching on the next frame boundary reached. The DMA channel will remain enabled for pattern matching until the MDMAENx bit is set to “0”. Once the MDMAENx bit is set to “0”, the DMA channel will be disabled on the next frame boundary reached.

MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)

The MXVR_INT_STAT_0 register indicates the current status of all events that can generate a Status Change Interrupt or a Control Message Interrupt in the MXVR. Each bit in the MXVR_INT_STAT_0 indicates whether a particular event has occurred. If the corresponding interrupt enable bit in the MXVR_INT_EN_0 is set to "1", the occurrence of that event will generate an interrupt.

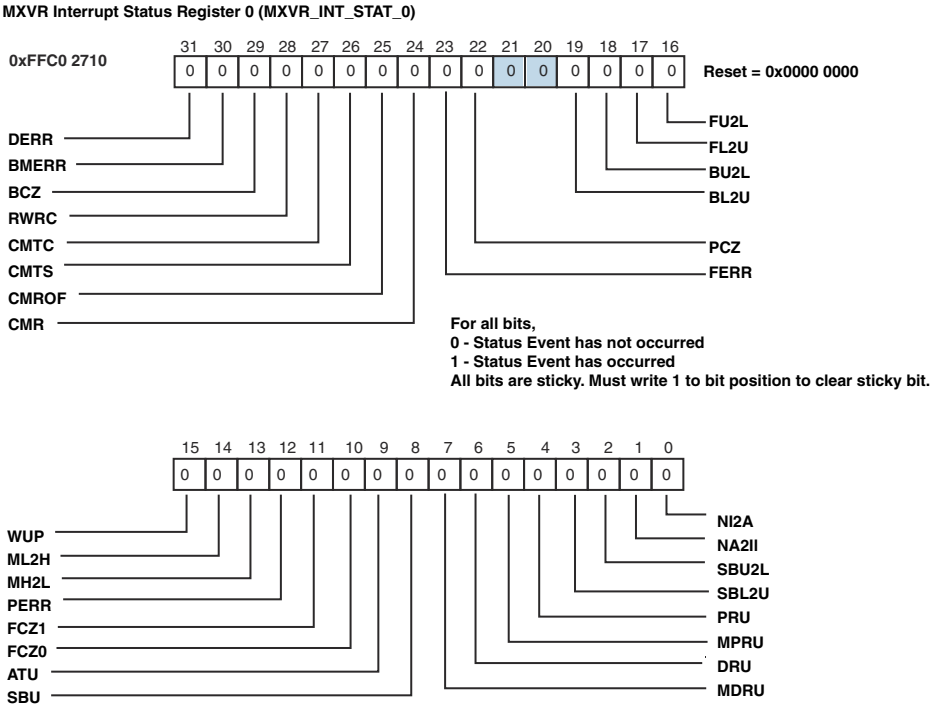


Figure 21-8. MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)

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, PCZ, FERR, BCZ, BMERR and DERR. The following status events generate the Asynchronous Packet Interrupt: APR, APROF, APTS, and APTC.

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 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 has been 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 has been received and processed or when the Allocation Table has been received over the network (once every 1024 frames). When in Slave Mode (MMSM = "0"), ATU will assert when the Allocation Table has been received over the network (once every 1024 frames). The MXVR does not determine whether the Allocation Table has changed—only that the Allocation Table has been received. Software must read the Allocation Table registers to determine if any changes have been made. If the ATUEN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of ATU will generate a Status Change Interrupt. The ATU bit can be cleared by writing a "1" to the ATU bit position. Note that it is recommended to only read the Allocation Table (MXVR_ALLOC_x registers) either immediately following an ATU event or immediately following a BCZ event to avoid the possibility of reading the Allocation Table while it is in the process of being updated.

The Parity Error (PERR) interrupt event will change to "1" whenever the calculated parity of the received frame does not match the parity bit in that frame. If the PERREN bit is set to "1" in the MXVR_INT_EN_1 register, the assertion of PERR will generate an Status Change Interrupt. The PERR bit can be cleared by writing a "1" to the PERR bit position.

The $\overline{\text{MRXON}}$ Low to High (ML2H) interrupt event will change to "1" when 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 $\overline{\text{MRXON}}$ High to Low (MH2L) interrupt event will change to "1" when 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 has been 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 has been started by writing a value to the MXVR_FRAME_CNT_0 register and Frame Counter 0 has decremented down to zero. If the FCZ0EN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of FCZ0 will generate an Status Change Interrupt. The FCZ0 bit can be cleared by writing a "1" to the FCZ0 bit position.

The Frame Counter 1 Zero (FCZ1) interrupt event will change to "1" when Frame Counter 1 has been started by writing a value to the MXVR_FRAME_CNT_1 register and Frame Counter 1 has decremented down to zero. If the FCZ1EN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of FCZ1 will generate an Status Change Interrupt. The FCZ1 bit can be cleared by writing a "1" to the FCZ1 bit position.

The Frame Unlocked to Locked (FU2L) interrupt event will change to “1” when the Frame Locked (FLOCK) bit in the MXVR_STATE_0 register changes from Frame Unlocked (FLOCK=”0”) to Frame Locked (FLOCK=”0”). If the FU2LEN bit is set to “1” in the MXVR_INT_EN_0 register, the assertion of the FU2L will generate a Status Change Interrupt. The FU2L bit can be cleared by writing a “1” to the FU2L bit position.

The Frame Locked to Unlocked (FL2U) interrupt event will change to “1” when the Frame Locked (FLOCK) bit in the MXVR_STATE_0 register changes from Frame Locked (FLOCK=”1”) to Frame Unlocked (FLOCK=”0”). If the FL2UEN bit is set to “1” in the MXVR_INT_EN_0 register, the assertion of the FL2U will generate a Status Change Interrupt. The FL2U bit can be cleared by writing a “1” to the FL2U bit position.

The Block Unlocked to Locked (BU2L) interrupt event will change to “1” when the Block Locked (BLOCK) bit in the MXVR_STATE_0 register changes from Block Unlocked (BLOCK=”0”) to Block Locked (BLOCK=”0”). If the BU2LEN bit is set to “1” in the MXVR_INT_EN_0 register, the assertion of the BU2L will generate a Status Change Interrupt. The BU2L bit can be cleared by writing a “1” to the BU2L bit position.

The Block Locked to Unlocked (BL2U) interrupt event will change to “1” when the Block Locked (BLOCK) bit in the MXVR_STATE_0 register changes from Block Locked (BLOCK=”1”) to Block Unlocked (BLOCK=”0”). If the BL2UEN bit is set to “1” in the MXVR_INT_EN_0 register, the assertion of the BL2U will generate a Status Change Interrupt. The BL2U bit can be cleared by writing a “1” to the BL2U bit position.

The PLL Counter Zero (PCZ) interrupt event will change to “1” when the PLL Counter has been started by writing a value to the PLLCNT field and setting the PLLCNTEN bit to “1” in the MXVR_PLL_CTL_1 register and the PLL Counter has decremented down to zero. If the PCZEN bit is set to 1 in the MXVR_INT_EN_0, the assertion of PCZ will generate a Status Change Interrupt. The PCZ bit can be cleared by writing a “1” to the PCZ bit position.

The FIFO Error (FERR) interrupt event will change to “1” when one of the MXVR internal FIFO’s overflows or underflows. This condition will most likely cause data corruption. This is a catastrophic event and the MXVR will automatically disable the effected transmit DMA channels. The internal FIFO underflows and overflows occur when the MXVR DMA channels cannot get enough internal DMA bus bandwidth for transfers to and from L1 to support the network interface. This normally would only happen if the system clock and/or the core clock frequency are lowered to a point where the internal busses cannot provide enough bandwidth to support all the enabled peripherals. If the FERREN bit is set to “1“in the MXVR_INT_EN_0 register, the assertion of FERR will generate a Status Change Interrupt. The FERR bit can be cleared by writing a “1” to the FERR bit position. If the FERR occurs due to the Synchronous Transmit FIFO underflowing all Synchronous Data Transmit DMA channels will automatically be disabled. If the FERR occurs due to the Asynchronous Packet Transmit FIFO underflowing, the Asynchronous Packet Transmit DMA channel will be disabled. If the FERR occurs due to the Synchronous Receive FIFO or the Asynchronous Packet Receive FIFO overflowing, the associated DMA channels will not automatically be disabled; however, the data received should be assumed to be corrupted.

If the FERR event ever occurs when running an application, the application code should be changed (the system clock and/or core clock frequency should be increased or the amount of DMA bandwidth being used should be decreased). The FERR event should never be allowed to occur in an application as this event indicates that data corruption may be occurring. In addition, if the FERR event occurs the MXVR must be disabled and re-enabled in order to reset the internal FIFO’s prior to re-enabling DMA channels.

The Control Message Received (CMR) interrupt event will change to "1" once a complete control message has been received by the MXVR and stored into the Control Message Receive Buffer. The CMR bit will not be set for System control messages or Normal control messages that fail the CRC check. If the CMREN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of CMR will generate a Control Message Interrupt. The CMR bit can be cleared by writing a "1" to the CMR bit position.

The Control Message Receive Buffer Overflow (CMROF) interrupt event will change to "1" when the Control Message Receive Buffer is full and a new control message is received over the network by the MXVR. The control message that is received by the MXVR when the Control Message Receive Buffer is full will be completely lost (the MXVR will respond with “Buffer Full” transmission status). If the CMROFEN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of CMROF will generate a Control Message Interrupt. The CMROF bit can be cleared by writing a "1" to the CMROF bit position.

The Control Message Transmit Buffer Successfully Sent (CMTS) interrupt event will change to "1" when the complete control message in the Control Message Transmit Buffer has been transmitted and the Transmission Status received back after the message has circled the network has been 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 has been cancelled. The transmission of the control message can only be cancelled while the MXVR is arbitrating for the control message channel. Once the MXVR has won arbitration, the transmission cannot be cancelled. If the CMTCEN

bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of CMTC will generate a Control Message Interrupt. The CMTC bit can be cleared by writing a "1" to the CMTC bit position.

The Remote Write Control Message Complete (RWRC) interrupt event will change to "1" when an incoming Remote Write Control Message has been processed and the received data has been 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 has been written to by another node.

The Block Counter Zero (BCZ) interrupt event will change to "1" when the Block Counter has been started by writing a value to the MXVR_BLOCK_CNT register and Block Counter has decremented down to zero. If the BCZEN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of BCZ will generate an Status Change Interrupt. The BCZ bit can be cleared by writing a "1" to the BCZ bit position. Note that the Block Counter only decrements at the beginning of Normal Blocks and not on the blocks containing the Allocation Table.

The Biphase Mark Coding Error (BMERR) interrupt event will change to "1" when there is a biphase mark code violation in any part of the frame other than the expected code violations in the preambles. If the BMERREN bit is set to "1" in the MXVR_INT_EN_0 register, the assertion of BMERR will generate a Status Change Interrupt. The BMERR bit can be cleared by writing a "1" to the BMERR bit position.

The DMA Error (DERR) interrupt event will change to "1" when one of the DMA channels encounters an error. DMA errors occur when the DMA channel attempts to access an illegal address in L1 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. 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.

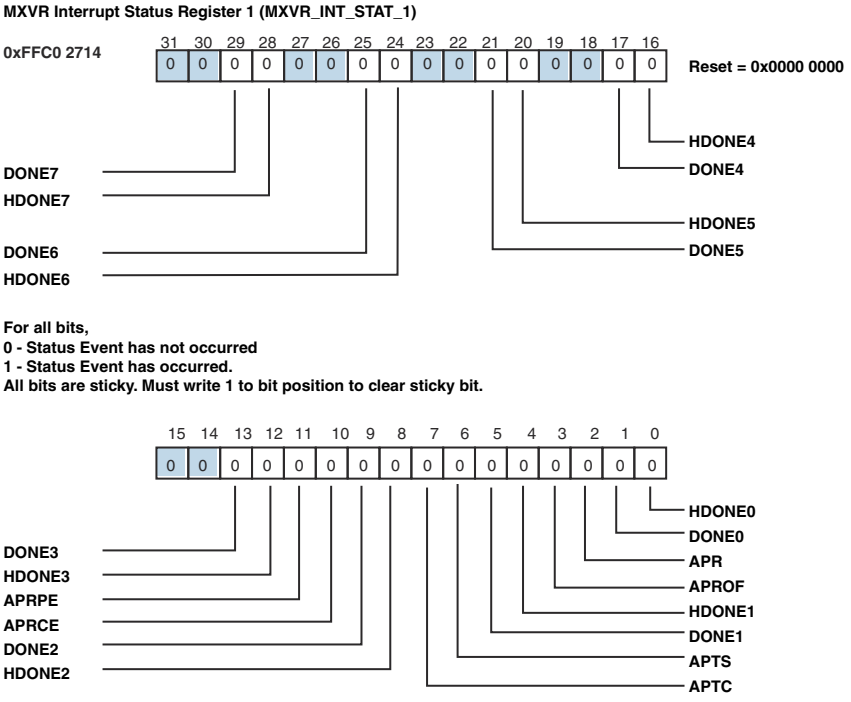


Figure 21-9. MXVR Interrupt Status Register_1 (MXVR_INT_STAT_1)

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 has been received by the MXVR and stored into the Asynchronous Packet Receive Buffer. If the APREN bit is set to "1" in the MXVR_INT_EN_1 register, the assertion of APR will generate an Asynchronous Packet Interrupt. The APR bit can be cleared by writing a "1" to the APR bit position.

The Asynchronous Packet Receive Buffer Overflow (APROF) interrupt event will change to "1" when the Asynchronous Packet Receive Buffer is full and a new asynchronous packet is received over the network by the MXVR. The asynchronous packet that is received by the MXVR when the Asynchronous Packet Receive Buffer is full will be completely lost. If the APROFEN bit is set to "1" in the MXVR_INT_EN_1 register, the assertion of APROF will generate an Asynchronous Packet Interrupt. The APROF bit can be cleared by writing a "1" to the APROF bit position.

The Asynchronous Packet Transmit Buffer Successfully Sent (APTS) interrupt event will change to "1" when the complete asynchronous packet in the Asynchronous Packet Transmit Buffer has been transmitted. If the APTSEN bit is set to "1" in the MXVR_INT_EN_1 register, the assertion of APTS will generate an Asynchronous Packet Interrupt. The APTS bit can be cleared by writing a "1" to the APTS bit position.

The Asynchronous Packet Transmit Buffer Successfully Cancelled (APTC) interrupt event will change to "1" when the transmission of the asynchronous packet in the Asynchronous Packet Transmit Buffer has been 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 has been 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 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.

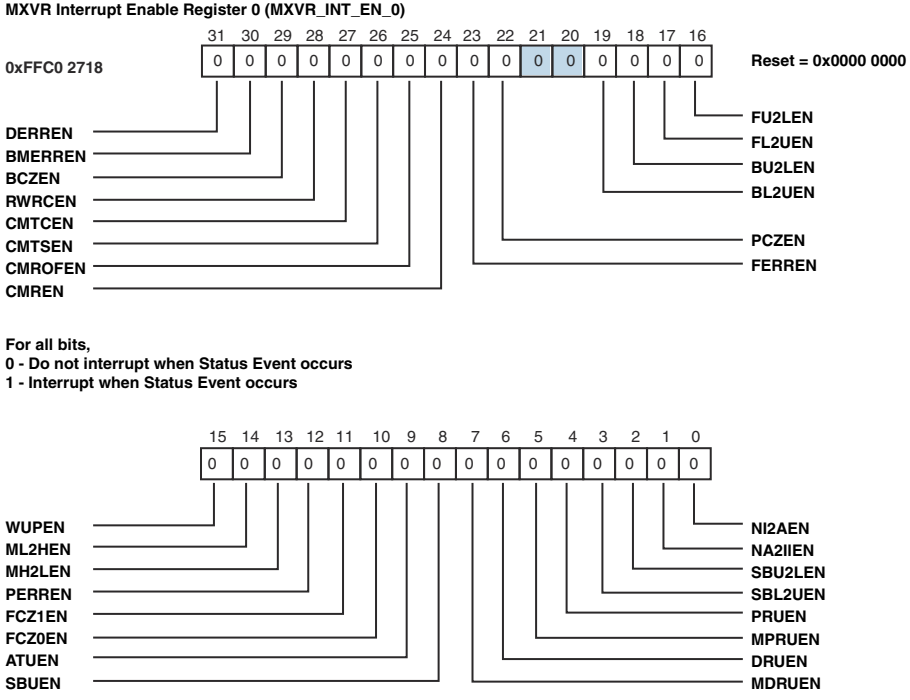


Figure 21-10. MXVR Interrupt Enable Register 0 (MXVR_INT_EN_0)

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)
- 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)
- $\overline{\text{MRXON}}$ High to Low interrupt enable (MH2LEN)
- $\overline{\text{MRXON}}$ 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)
- PLL Counter Zero interrupt enable (PCZEN)
- FIFO Error interrupt enable (FERREN)
- Frame Counter 0 Zero interrupt enable (FCZ0EN)
- Frame Counter 1 Zero interrupt enable (FCZ1EN)
- Control Message Received interrupt enable (CMREN)
- Control Message Receive Buffer Overflow interrupt enable (CMROFEN)
- 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.

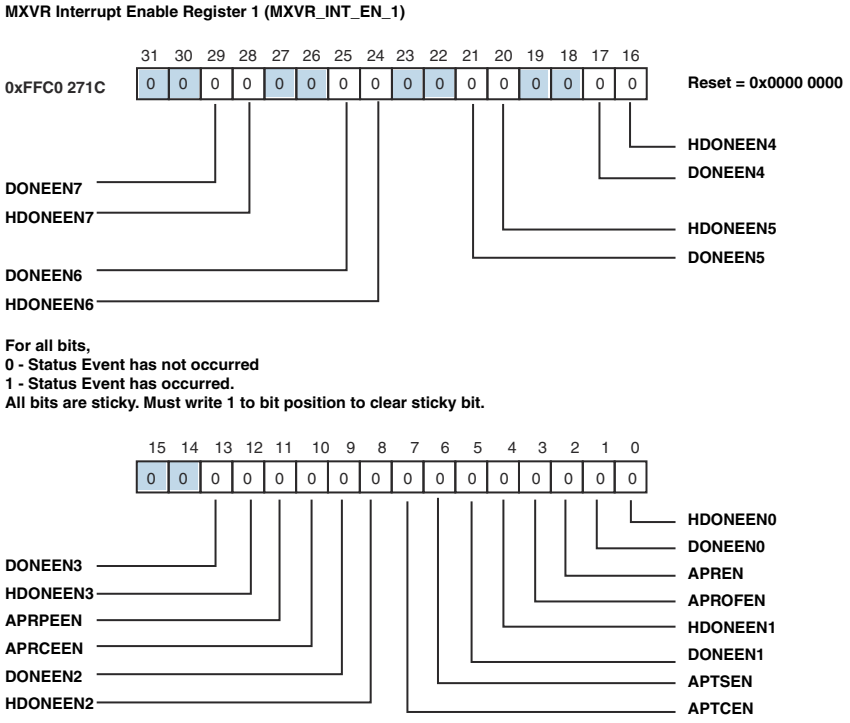


Figure 21-11. MXVR Interrupt Enable Register 1 (MXVR_INT_EN_1)

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.

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.

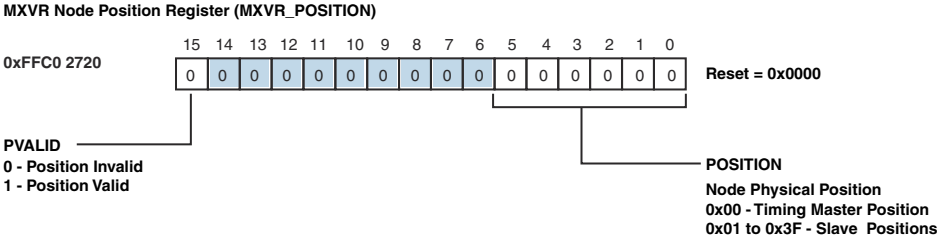


Figure 21-12. MXVR Node Position Register (MXVR_POSITION)

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 has been 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 (i.e. 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).

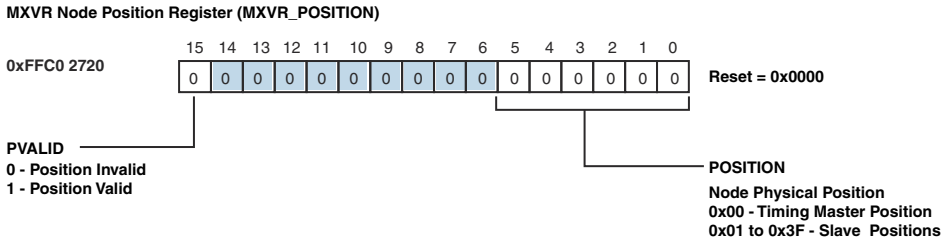


Figure 21-13. MXVR Maximum Node Position Register (MXVR_MAX_POSITION)

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 (i.e. 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.

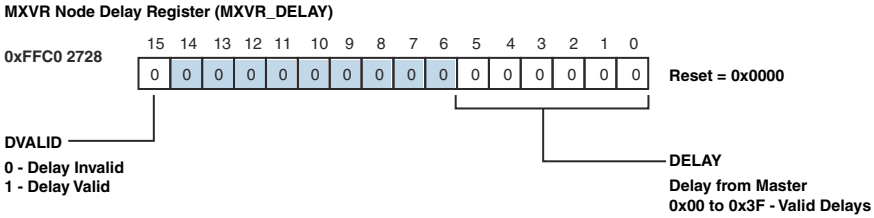


Figure 21-14. MXVR Node Frame Delay Register (MXVR_DELAY)

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 has been correctly determined, the DVALID bit will change to a "1" and the DELAY field will contain the node delay value. Subsequent changes to the node delay (i.e. 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 memory is not frame delayed in the process of transferring the data and synchronous data that is DMA'ed from L1 memory to the MXVR for transmit is not frame delayed in the process of transferring the data.

The actual time delay of data transmitted from L1 memory of one MXVR node "A" to the L1 memory of MXVR node "B" can be calculated using one of three formulas:

If (POSITIONA < POSITIONB),

t_{delay} = 2 * (DELAYA - DELAYB) * (1 / Fs)

If (POSITIONA > POSITIONB) and (SDELAYA ==”0”),

t_{delay} = 2 * (MDELAY - DELAYA + DELAYB) * (1 / Fs)

If (POSITIONA > POSITIONB) and (SDELAYA ==”1”),

t_{delay} = 2 * (MDELAY - DELAYA + DELAYB - 1) * (1 / Fs)

MXVR Maximum Node Frame Delay 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.

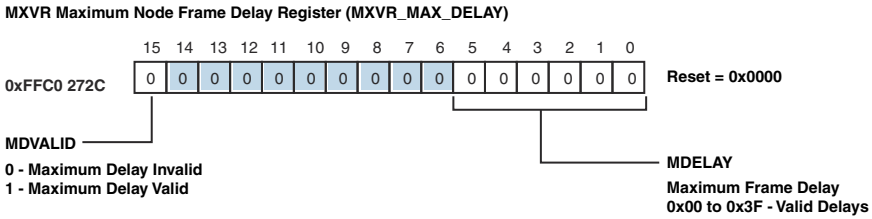


Figure 21-15. MXVR Maximum Node Frame Delay Register (MXVR_MAX_DELAY)

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 has been correctly determined, the MDVALID bit will change to a "1" and the MDELAY field will contain the total number of node delays. Then the Master will distribute the total number of delays in the network to the Slave nodes once every 1024 frames.

When the MXVR is enabled in Slave Mode, the MDVALID bit will be "0" until the MXVR has reached a lock level at which the total number of node delays can correctly received from the Master. Once the total number of node delays has been 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 (i.e. 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 0x0300 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.

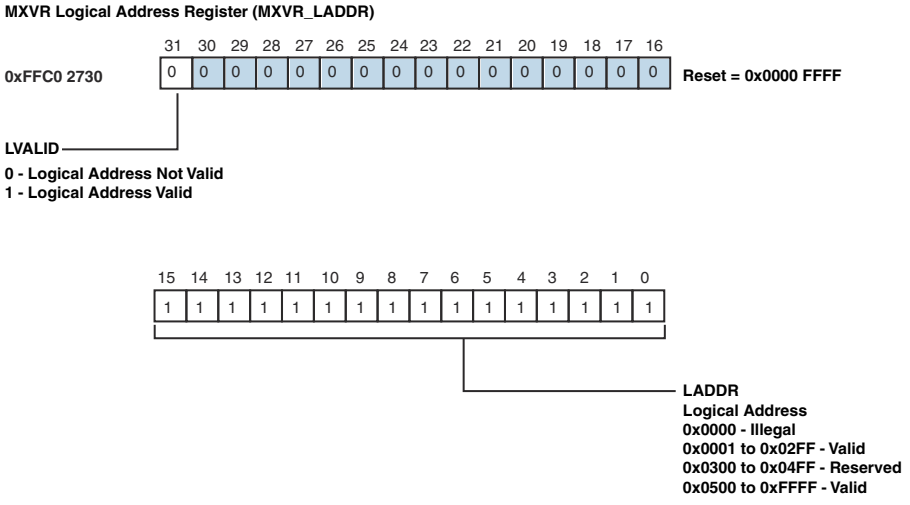


Figure 21-16. MXVR Logical Address Register (MXVR_LADDR)

There is an LVALID bit which should be written to a "1" once the LADDR field has been 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 has been 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.

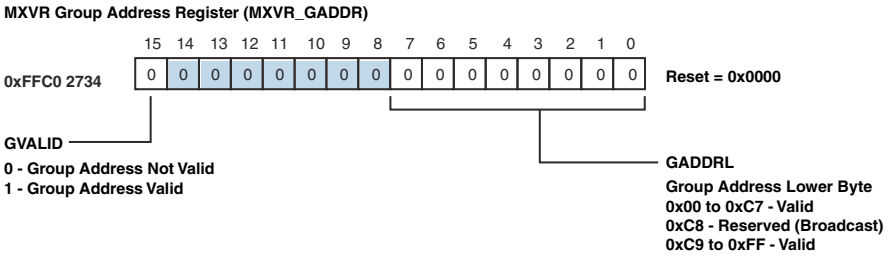


Figure 21-17. MXVR Group Address Register (MXVR_GADDR)

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 has been 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.

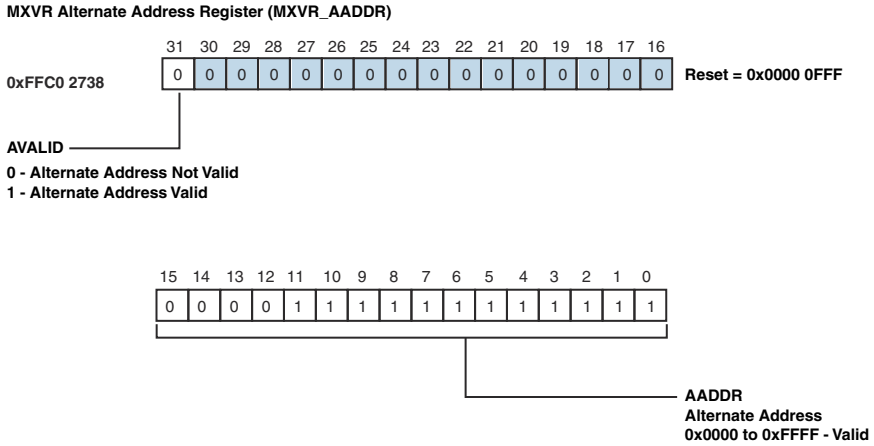


Figure 21-18. MXVR Alternate Address Register (MXVR_AADDR)

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).

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 21-19 shows MXVR_ALLOC_0 register as an example of one of the Allocation Table registers. All other Allocation Table registers have the same format.

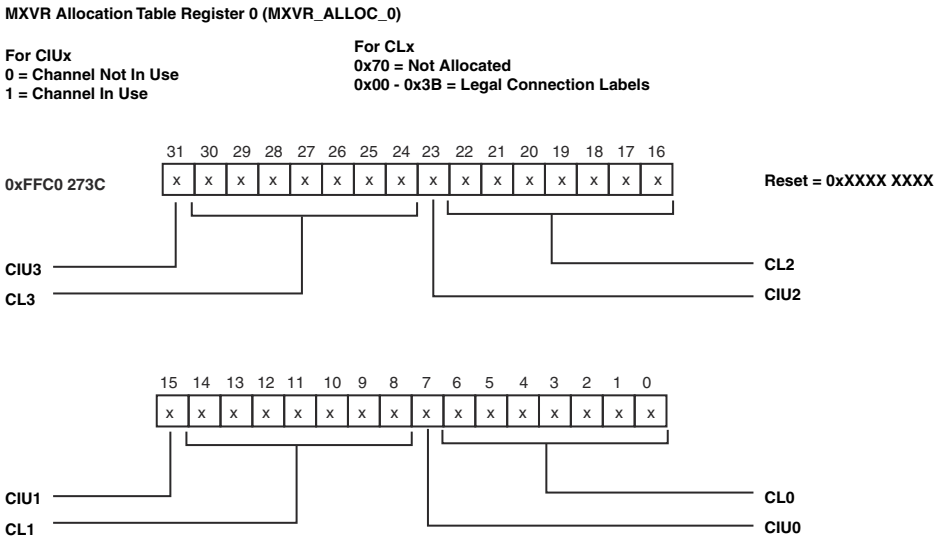


Figure 21-19. MXVR Allocation Table Register (MXVR_ALLOC_0)

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 has been allocated and is associated with all other physical channels which have the same CLx field value.

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

The Master node modifies its Allocation Table based on 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 CIUx bit for each physical channel that Slave node is using. In this way, once the Allocation Table returns back to the Master, the Master's Allocation Table will show which channels are "In-Use" for the entire network. Note that in each Slave node, the CIUx bits only reflect which channels are "In-Use" by upstream nodes (nodes with lower POSITION values).

MXVR Synchronous Logical Channel Assignment 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 21-20 shows MXVR_SYNC_LCHAN_0 register as an example of one of the Synchronous Logical Channel Assignment registers. All other Synchronous Logical Channel Assignment registers have the same format.

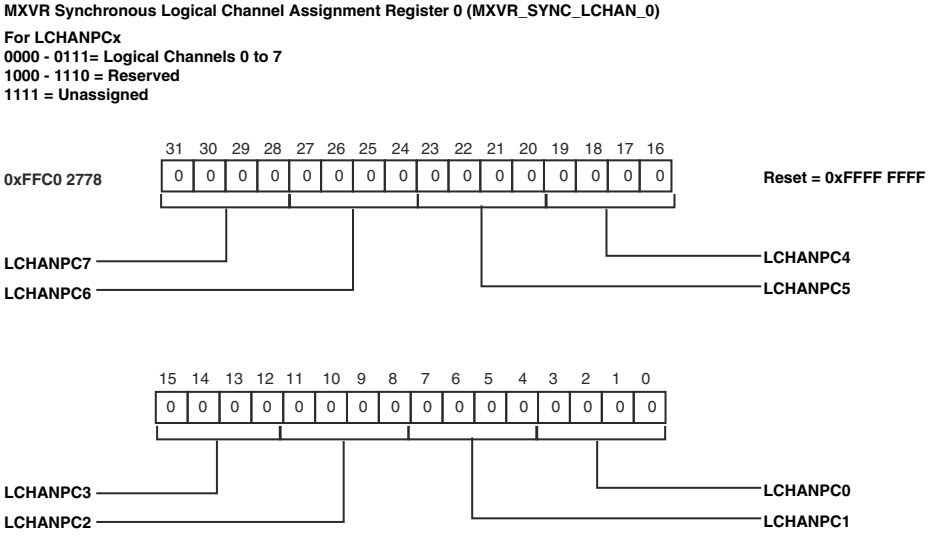


Figure 21-20. MXVR Synchronous Logical Channel Assignment Register (MXVR_SYNC_LCHAN_0)

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 has been programmed to receive data from logical channel 6 (LCHAN3=b#0110), then Synchronous Data DMA Channel 3 will DMA data received on physical channels 5, 8, and 30 into L1 memory.

Note that the logical channel numbers assigned to the LCHANPCx fields have no meaning other than to associate physical channels with each other and assign them to DMA channels. These logical channel numbers are completely independent of the Connection Label numbers in the Allocation Table.

MXVR DMA Channel x 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 memory or can transfer synchronous data stored in L1 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 memory or from L1 memory to any set of transmitted channels.

The DMA channel is enabled by setting the DMAx Enable (MDMAENx) bit to "1" or disabled by setting the MDMAEN bit to "0". When the MDMAENx bit is set to "1", the MXVR_DMAx_START_ADDR and MXVR_DMAx_COUNT registers should not be written. In addition when the MDMAENx bit is set to "1", all bits in the MXVR_DMAx_CONFIG register except for the MDMAENx bit will be read-only and writes to other bits in the MXVR_DMAx_CONFIG register will have no effect.

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 memory buffer. When the DDx bit is set to "0", the DMA channel will transfer data from an L1 memory buffer to the MXVR to be transmitted.

The DMAx Logical Channel (LCHANx) field determines which logical channel in the incoming frame will be received and DMA'd to L1 memory or which logical channel in the outgoing frame will be DMA'd from L1 memory and transmitted. The logical channels are defined in the MXVR_SYNC_LCHAN_x registers. Two DMA channels can have the same LCHANx field set as long as the data direction for the two channels is different (one for receive, one for transmit). Programming more than one DMA channel with the same data direction and the same LCHANx value is illegal.

The DMAx Bit-Swap Enable (BITSWAPENx) bit enables or disables bit swapping of the data that is DMA'd to and from L1 memory. If BITSWAPENx is set to "1", the data bits will be swapped on a byte-wise basis as follows:

bit 7 => bit 0 and bit 0 => bit 7

bit 6 => bit 1 and bit 1 => bit 6

bit 5 => bit 2 and bit 2 => bit 5

bit 4 => bit 3 and bit 3 => bit 4

For example, data value 0x35 when bit-swapped becomes 0xAC. If BITSWAPEN is set to "0", no bit swapping will take place. Note that bit-swapping and byte-swapping may be used in conjunction.

The DMAx Byte-Swap Enable (BYSWAPENx) bit enables or disables byte swapping of the data that is DMA'd to and from L1 memory. If BYSWAPENx is set to "1", the data byte 0 will be swapped with data byte 1. If BYSWAPENx is set to "0", no byte swapping will take place. For example, data value 0x3586 when byte-swapped becomes 0x8635. Byte swapping is done by reading and writing the L1 memory in a different order if byte swapping is enabled. For example, normally data will be read/written from/to L1 in the following address order: 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, etc. If byte swapping is enabled, data will be read/written from/to L1 in the following address order: 0x01, 0x00, 0x03, 0x02, 0x05, 0x04, etc. Note that when byte swapping is enabled, the MXVR_DMAx_CURR_ADDR

will reflect the normal address incrementing (0x00, 0x01, 0x02, 0x03, etc.) even though the L1 memory accesses will be occurring in the byte swapping address order. Note that bit-swapping and byte-swapping may be used in conjunction.

The DMAx Operation Flow (MFLOWx) field determines the operating mode of the DMA channel. Each DMA channel can operate in Stop Mode, Autobuffer Mode, Synchronous Packet-Fixed Count Mode, Synchronous Packet-Variable Count Mode, and Synchronous Packet-Start/Stop Mode.

In Stop Mode, once the DMA is enabled a fixed number of bytes of data will be transferred from the logical channel to an L1 memory buffer (receive) or from an L1 memory buffer to the logical channel (transmit). The starting address of the L1 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 memory buffer (receive) or from an L1 memory buffer to the logical channel (transmit). The starting address of the L1 memory buffer is programmed in the MXVR_DMAx_START_ADDR register and the number of bytes to be transferred is programmed in the MXVR_DMAx_COUNT register. The DMA channel will set the HDONEx status event when half of the total number of bytes are completed, and will set the DONEx status event when the total number of bytes are completed. Once all the transfers are done the DMA will remain

enabled and will restart from the address specified in the MXVR_DMAx_START_ADDR register and with the transfer count in the MXVR_DMAx_COUNT register. Disabling the DMA channel manually when the DMA is programmed for Autobuffer Mode will halt the DMA transfers and the values in the MXVR_DMAx_CURR_ADDR and MXVR_DMAx_CURR_COUNT will indicate where the DMA channel stopped. However, when the channel is re-enabled, the current address and count will reset back to the values programmed into the MXVR_DMAx_START_ADDR and MXVR_DMAx_COUNT.

The DMA channels have three Synchronous Packet Autobuffer Modes which allow the DMA channels to receive packetized data over the synchronous data channels. The three modes are Synchronous Packet-Variable Count Mode, Synchronous Packet-Start/Stop Mode, and Synchronous Packet-Fixed Count Mode. These DMA modes are only used when the MXVR is receiving data and the DMA channel is writing the data to L1 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 datastream 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 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 datastream. The position of the transfer count with respect to

the “start pattern” is programmed in the COUNTPOSx field. The MXVR_DMAx_CURR_COUNT will initially be set to 0xFFFF when the "start pattern" is found and will decrement with every transfer done prior to receiving the transfer count. Once the transfer count is received, the MXVR_DMAx_CURR_COUNT will be based on transfer count from the datastream. Once the DMA channel transfers the number of bytes based on the transfer count to L1 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 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 datastream 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

memory and at the same time will search for the “stop pattern” in the logical channel datastream. The “stop pattern” is selected by the STOPPATx field. The MXVR_DMAx_CURR_COUNT will initially be set to 0xFFFF when the "start pattern" is found and will decrement with every transfer done prior to receiving “stop pattern”. Once the DMA channel receives the “stop pattern”, the DMA will stop transferring data. The DMA channel will then repeat the process and start looking for the “start pattern” again.

The first packet (and subsequent odd packet numbers) of data received will be written to the address specified in the MXVR_DMAx_START_ADDR. The DMA transfers will continue until the "stop pattern" is found. Once the "stop pattern" is found, the HDONEx bit in the MXVR_INT_STAT_1 register will be set to "1". The second packet (and subsequent even packet numbers) of data received will be written to an address that is defined by the MXVR_DMAx_START_ADDR plus the value programmed in the MXVR_DMAx_COUNT. The DMA transfers will continue until the "stop pattern" is found. Once the "stop pattern" is found, the DONEx bit in the MXVR_INT_STAT_1 register will be set to "1". Received packets will ping-pong between these two L1 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 datastream 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 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 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 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 "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 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 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" has been found when operating in Synchronous Packet-Variable Count Mode. The COUNTPOSx indicates the position of the transfer count by giving the number of bytes between the last byte of the "start pattern" to the first byte of the transfer count. The COUNTPOSx can range from 0 bytes after the end of the "start pattern" to 7 bytes after the end of the "start pattern." For example, if the COUNTPOSx was set to 0, then the transfer count would be found in the first two bytes in the logical channel after the end of the “start pattern”. If the COUNTPOSx was set to 7, then the transfer count would be found in the eight and ninth bytes in the logical channel after the end of the "start pattern". The most significant byte of the transfer count is received first, and followed by the least significant byte of the transfer count. Note that the number of bytes set by the COUNTPOSx field is with respect to the logical channel datastream. In other words, the “start pattern” and transfer count will be in the same logical channel but may be in different frames. Note that the bytes of data between the “start pattern” and the transfer count, and the transfer count itself will be DMA’ed to L1 memory.

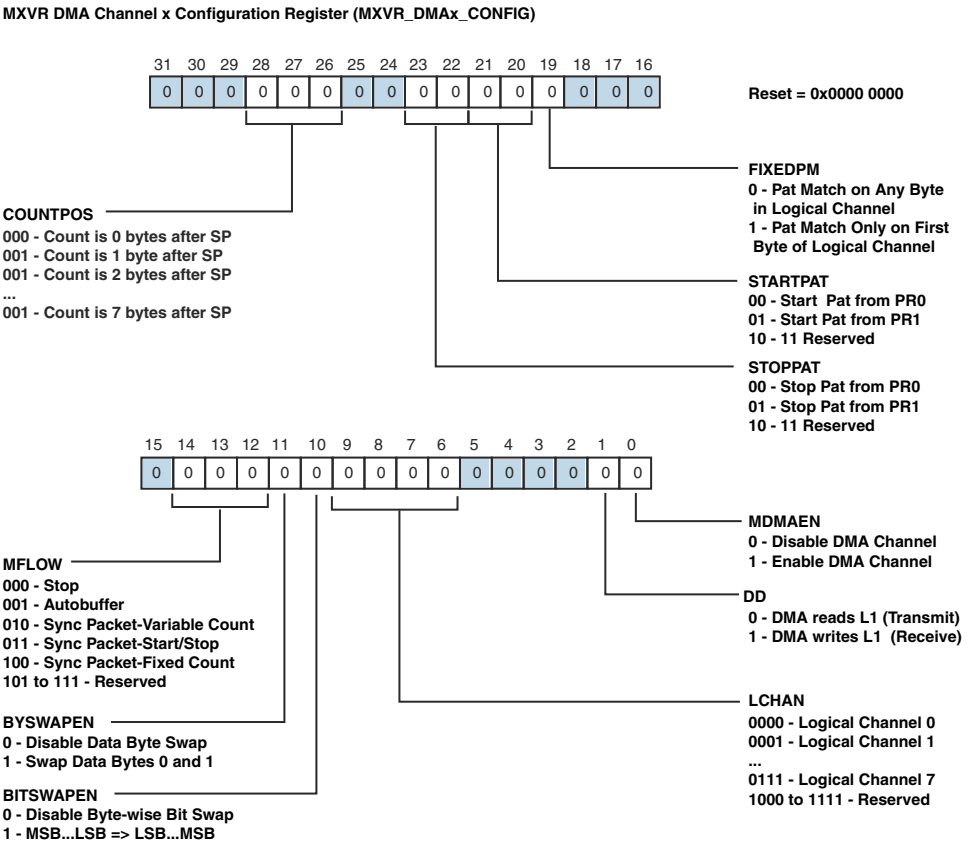


Figure 21-21. MXVR DMA Channel x Configuration Register (MXVR_DMAx_CONFIG)

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 memory. Therefore, bits 31-24 are fixed to 0xFF.

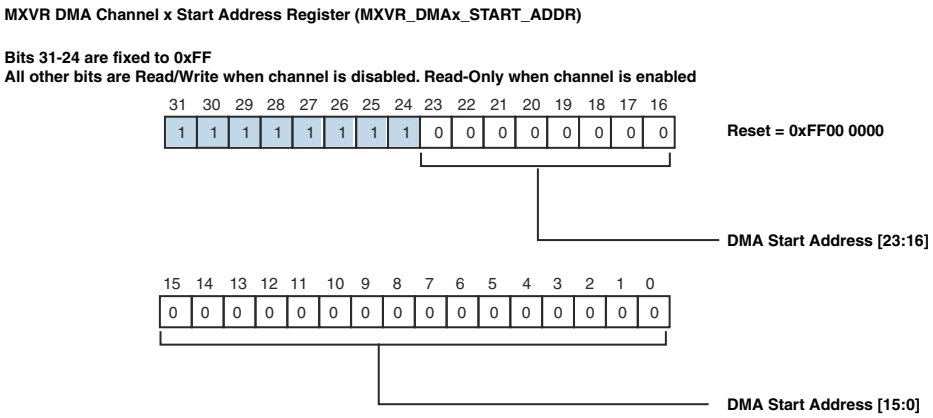


Figure 21-22. MXVR DMA Channel x Start Address Registers (MXVR_DMAx_START_ADDR)

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 has been selected with the MFLOWx field:

If the DMA is operating in Stop Mode, once all the transfers specified in the corresponding MXVR_DMAx_COUNT have been done, the DMA will automatically disable.

If the DMA channel is operating in Autobuffer Mode, once all of the transfers specified in the corresponding MXVR_DMAx_COUNT have been done, the DMA will then jump back to the start address programmed in the MXVR_DMAx_START_ADDR and DMA operations will continue from there. In this way the data received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT/2.

If the DMA channel is operating in Synchronous Packet-Variable Count Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the transfer count field in the packet itself have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the transfer count field in the packet itself have been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

If the DMA channel is operating in Synchronous Packet-Start/Stop Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the amount of data received between the "start pattern" and the "stop pattern" have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the amount of data received between the "start pattern" and the "stop pattern" have been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

If the DMA channel is operating in Synchronous Packet-Fixed Count Mode, the first packet received will be written to the MXVR_DMAx_START_ADDR. Once all of the transfers specified by the fixed

count in the MXVR_DMAx_COUNT have been done, the second packet received will be written to MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. Once all of the transfers specified by the fixed count in the MXVR_DMAx_COUNT have been done, the third packet received will be written to MXVR_DMAx_START_ADDR. In this way the packets received will alternate between being written to the first memory buffer at MXVR_DMAx_START_ADDR and the second memory buffer at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

MXVR DMA Channel x Current Address 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 memory. Therefore, bits 31–24 are fixed to 0xFF. 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 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 memory. The maximum

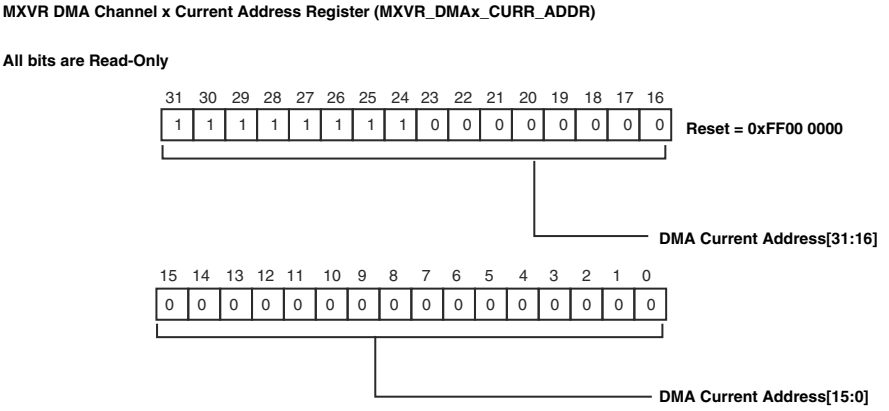


Figure 21-23. MXVR DMA Channel x Current Address Register (MXVR_DMAx_CURR_ADDR)

MXVR_DMAx_COUNT value is 65535 (giving a maximum data block size to be DMA'd of 64 kbytes). The value 0x0000 is illegal and should not be written to the MXVR_DMAx_COUNT register.

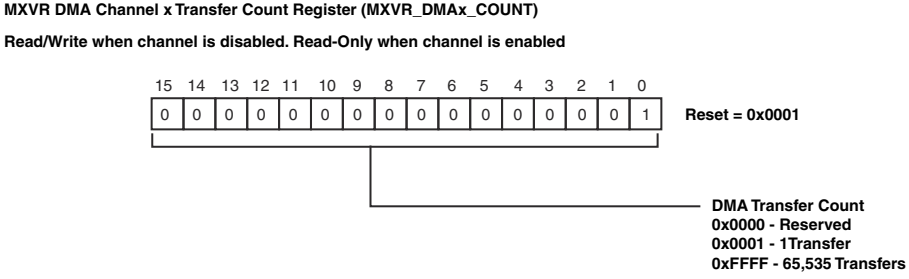


Figure 21-24. MXVR DMA Channel x Transfer Count Registers (MXVR_DMAx_COUNT)

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 has been selected with the MFLOWx field:

If the DMA is operating in Stop Mode, the MXVR_DMAx_COUNT value is the total number of bytes to be transferred. Once half of the transfers specified have been completed, the HDONE interrupt event will be generated. Once all the transfers specified have completed, the DONE interrupt event will be generated and the DMA will automatically be disabled.

If the DMA channel is operating in Autobuffer Mode, the MXVR_DMAx_COUNT value is the total number of bytes to be transferred before the address is reset back to the MXVR_DMAx_START_ADDR. Once half of the transfers specified have completed, the HDONE interrupt event will be generated. Once all the transfers specified have completed, the DONE interrupt event will be generated and the DMA will jump back to the start address programmed in the MXVR_DMAx_START_ADDR and DMA operations will continue from there.

If the DMA channel is operating in Synchronous Packet-Variable Count Mode, the MXVR_DMAx_COUNT value is the offset from the MXVR_DMAx_START_ADDR where every other packet will be written to. The first packet (third packet, fifth packet, seventh packet, etc.) received will be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The transfer count for each packet is included in the packet itself. Therefore, the MXVR_DMAx_COUNT value should be larger than the length of the largest packet to be received.

If the DMA channel is operating in Synchronous Packet-Start/Stop Mode, the MXVR_DMAx_COUNT value is the offset from the MXVR_DMAx_START_ADDR where every other packet will be written to. The first packet (third packet, fifth packet, seventh packet, etc.) received will be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at

MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT. The number of transfers to be done for each packet is determined by the packet itself based on the number of bytes between the “start pattern” and the “stop pattern”. Therefore, the MXVR_DMAx_COUNT value should be sufficiently large to hold the longest packet to be received.

If the DMA channel is operating in Synchronous Packet-Fixed Count Mode, the MXVR_DMAx_COUNT value is the number of bytes that will be transferred to store one packet. All packets will be of the same length. The first packet (third packet, fifth packet, seventh packet, etc.) received will be written starting at MXVR_DMAx_START_ADDR, while the second packet (fourth packet, sixth packet, eight packet, etc.) will be written starting at MXVR_DMAx_START_ADDR + MXVR_DMAx_COUNT.

MXVR DMA Channel x Current Transfer Count 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 has been selected with the MFLOWx field.

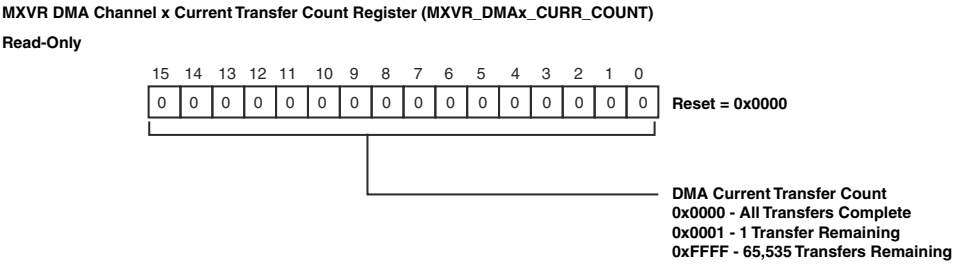


Figure 21-25. MXVR DMA Channel x Current Transfer Count Registers (MXVR_DMAx_CURR_COUNT)

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” has been found. Therefore, the MXVR_DMAx_CURR_COUNT will decrement from 0xFFFF and will stop when the “stop pattern” has been 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 (APTB) and an Asynchronous Packet Receive Buffer (APRB). The APRB is capable of holding two received Asynchronous Packets.

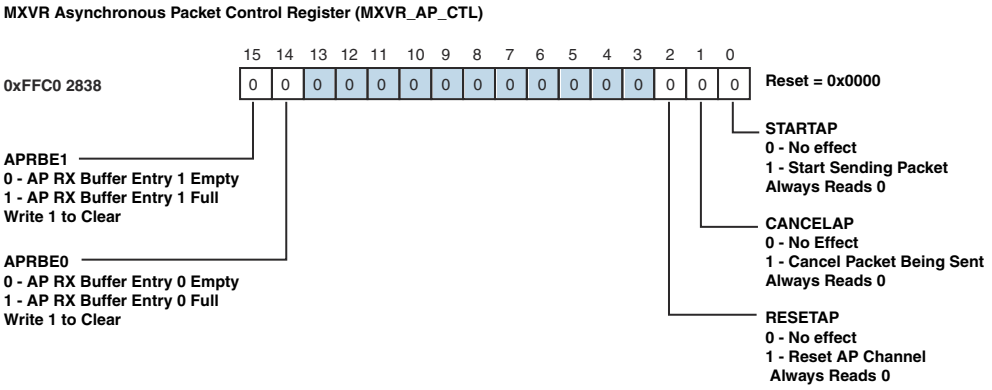


Figure 21-26. MXVR Asynchronous Packet Control Register (MXVR_AP_CTL)

The Start Asynchronous Packet Transmission (STARTAP) bit should be set to "1" once an asynchronous packet to be transmitted has been written to the APTB and the asynchronous packet is ready to be sent. Once the STARTAP bit is set to "1", arbitration for the asynchronous channel begins and continues until arbitration is won or until the asynchronous packet is cancelled with the CANCELAP bit. The STARTAP bit always reads as "0" and writing a "0" to the STARTAP bit has no effect.

The Cancel Asynchronous Packet Transmission (CANCELAP) bit allows an asynchronous packet transmission which is arbitrating for the asynchronous channel to be cancelled. Once arbitration has been 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 has been 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 memory. The APRB must be allocated 2048 bytes. The APRB can only reside in L1 memory and the APRB must be word aligned. Therefore, bits 31-24 are fixed to 0xFF and bit 0 is fixed to “0”.

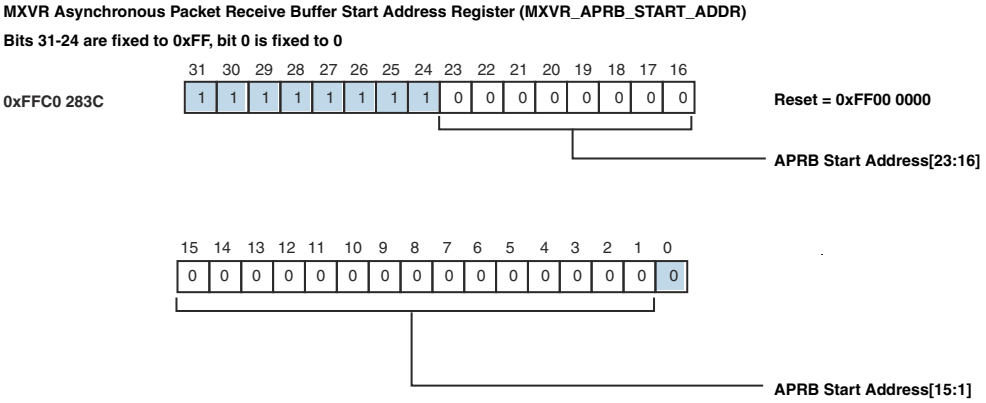


Figure 21-27. MXVR APRB Start Address Register (MXVR_APRB_START_ADDR)

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 memory. Therefore, bits 31-24 will always be 0xFF and bit 0 will always be “0”.

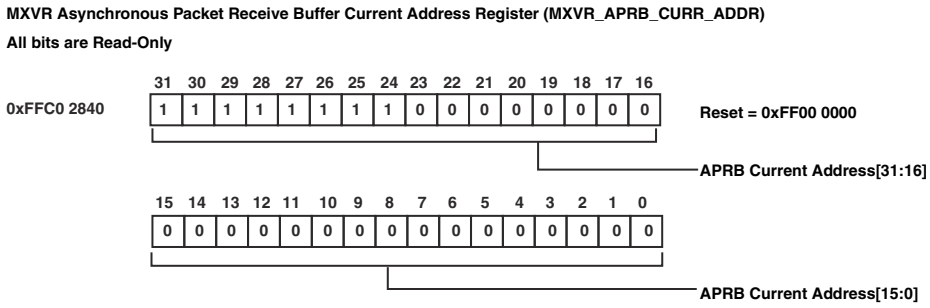


Figure 21-28. MXVR Asynchronous Packet Receive Buffer Current Address Register (MXVR_APRB_CURR_ADDR)

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 memory. Enough memory should be allocated to the APTB based on the largest packet to be transmitted. The APTB can only reside in L1 memory and the APTB must be word aligned. Therefore, bits 31-24 are fixed to 0xFF and bit 0 is fixed to "0".

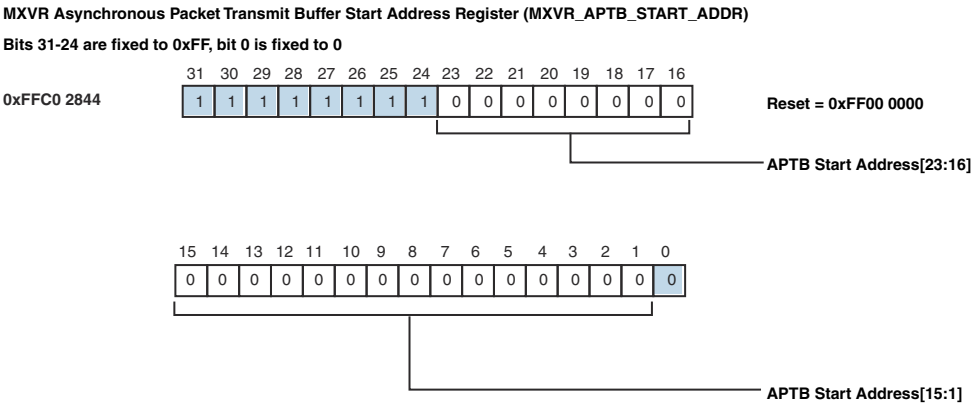


Figure 21-29. MXVR Asynchronous Packet Transmit Buffer Start Address Register (MXVR_APTB_START_ADDR)

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 memory. Therefore, bits 31-24 will always be 0xFF, and bit 0 will always be “0”.

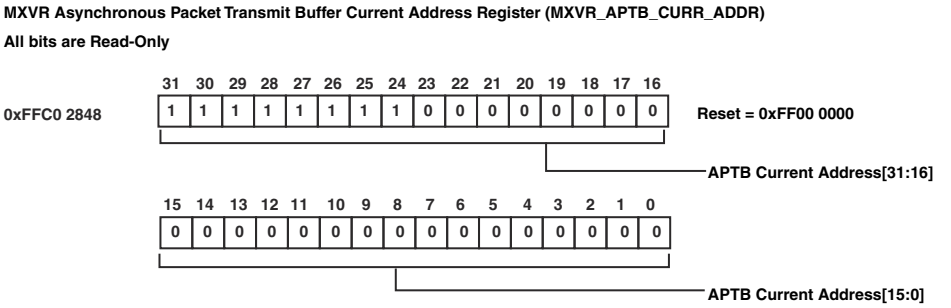


Figure 21-30. MXVR Asynchronous Packet Transmit Buffer Current Address Register (MXVR_APTB_CURR_ADDR)

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 Memory and holds one control message (system or normal) to be transmitted. The MXVR also uses a Control Message Receive Buffer (CMRB) which resides in L1 Memory and holds up to 16 received normal control messages.

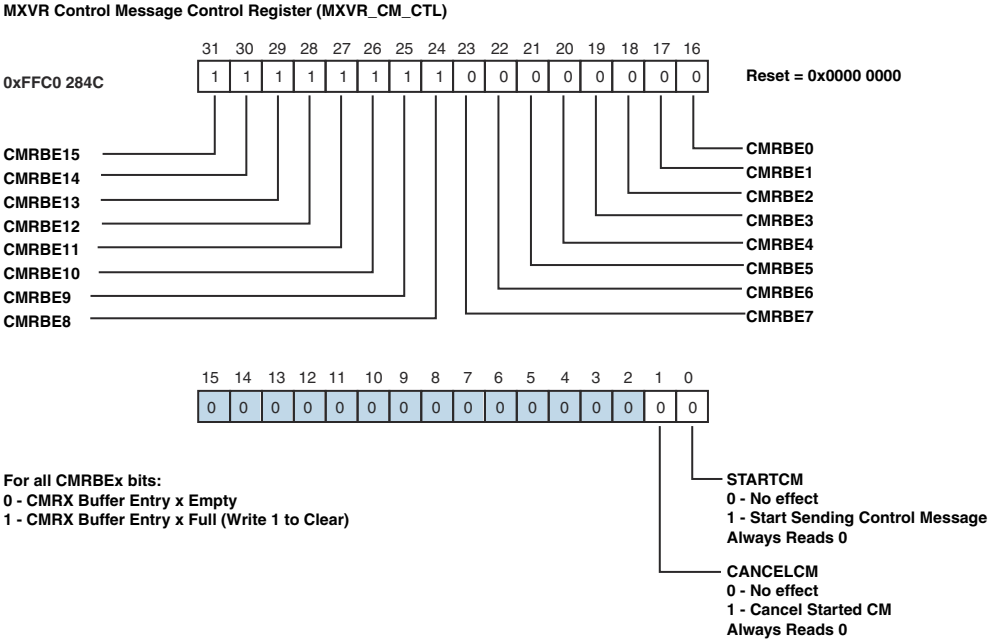


Figure 21-31. MXVR Control Message Control Register (MXVR_CM_CTL)

The Start Control Message Transmission (STARTCM) bit should be set to "1" when a control message has been 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 has been 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 has been 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 (CMRBEx) bits indicate whether entry x in the CMRB is full or empty. The CMRBEx bits are sticky bits which must be written with a "1" to clear. When a received normal control message is DMA'd to the CMRB entry, the corresponding CMRBEx bit will be set to "1". Once software has read the normal control message stored in that entry, a "1" should be written to the corresponding CMRBEx bit in order to clear the bit and to indicate that the entry is empty and can be used for another incoming normal control message. The MXVR will always attempt to DMA an incoming normal control message to the next sequential CMRB entry (first normal control message to CMRBE0, second to CMRBE1, ... , sixteenth to CMRBE15, seventeenth to CMRBE0, etc.). An overflow will occur if the next sequential CMRBEx bit is "1" when a new normal control message is arriving, and the CMRBOF bit in the MXVR_INT_STAT_0 register will be set to "1". In addition, when an overflow occurs the Transmission Status will be returned to the transmitter indicating "Receive Buffer Full".

MXVR Control Message Receive Buffer Start Address Register (MXVR_CMRB_START_ADDR)

The MXVR_CMRB_START_ADDR register sets the starting address for the Control Message Receive Buffer (CMRB) in L1 memory. The CMRB must be allocated 384 bytes. The CMRB can only reside in L1 memory and must be word aligned. Therefore, bits 31–24 are fixed to 0xFF and bit 0 is fixed to "0".

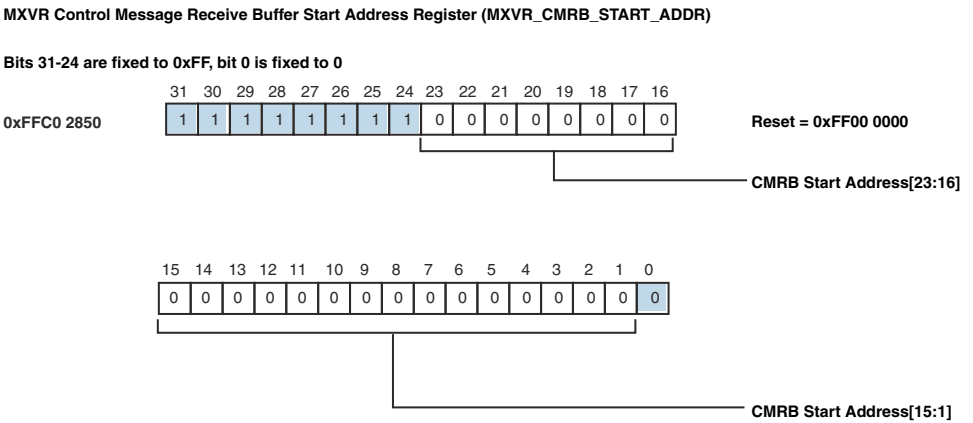


Figure 21-32. MXVR Control Message Receive Buffer Start Address Register (MXVR_CMRB_START_ADDR)

MXVR Control Message Receive Buffer Current Address Register (MXVR_CMRB_CURR_ADDR)

The MXVR_CMRB_CURR_ADDR register is a read-only register which gives the current address that the Normal Control Message Receive DMA channel is writing to in the CMRB. The CMRB can only reside in L1 memory and must be word aligned. Therefore, bits 31–24 will always be 0xFF and bit 0 will always be “0”.

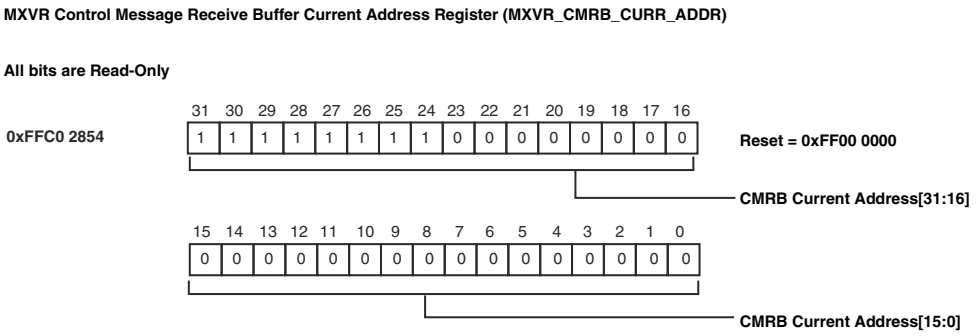


Figure 21-33. MXVR Control Message Receive Buffer Current Address Register (MXVR_CMRB_CURR_ADDR)

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 memory. The CMTB must be allocated 26 bytes. The CMTB can only reside in L1 memory and must be word aligned. Therefore, bits 31–24 are fixed to 0xFF and bit 0 is fixed to "0".

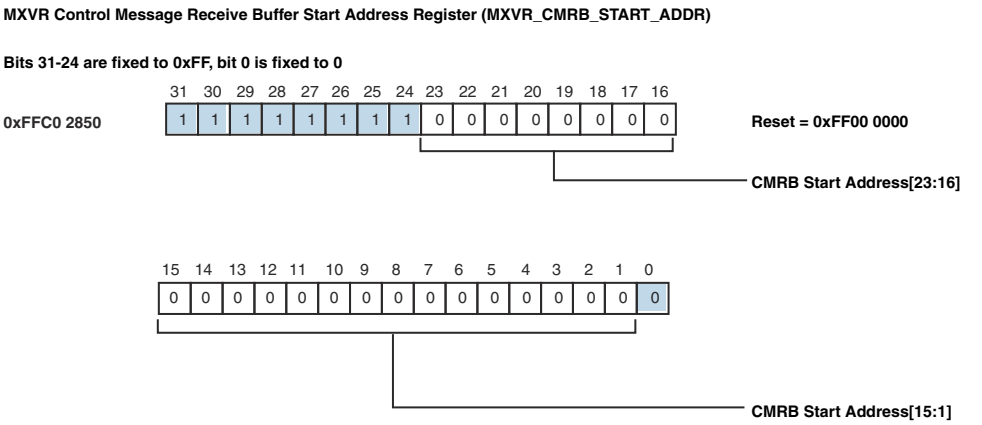


Figure 21-34. MXVR Control Message Transmit Buffer Start Address Registers (MXVR_CMTB_START_ADDR)

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 memory and must be word aligned. Therefore, bits 31–24 will always be 0xFF and bit 0 will always be “0”.

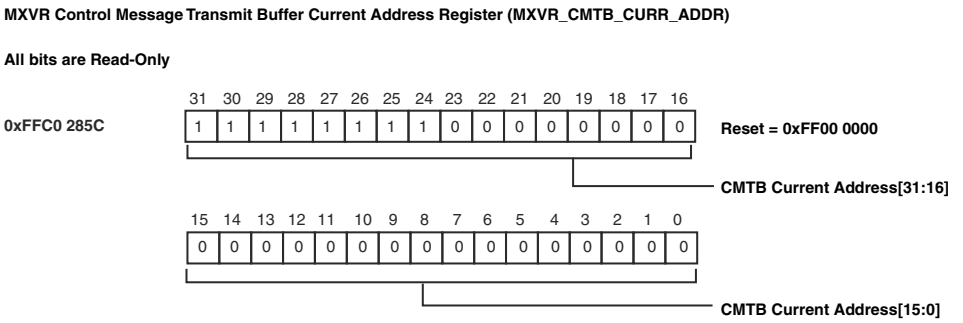


Figure 21-35. MXVR Control Message Transmit Buffer Current Address Register (MXVR_CMTB_CURR_ADDR)

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 memory. The RRDB must be allocated 258 bytes. The RRDB can only reside in L1 memory and must be word aligned. Therefore, bits 31–24 are fixed to 0xFF and bit 0 is fixed to "0".

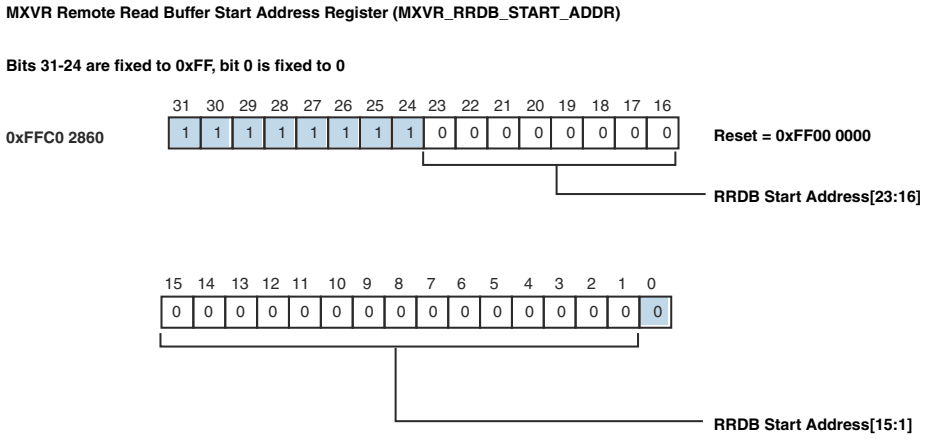


Figure 21-36. MXVR Remote Read Buffer Start Address Register (MXVR_RRDB_START_ADDR)

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 memory and must be word aligned. Therefore, bits 31–24 will always be 0xFF and bit 0 will always be “0”.

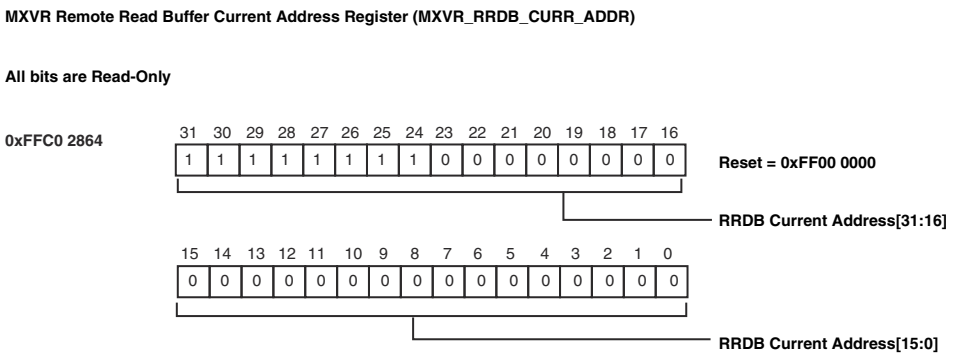


Figure 21-37. MXVR Remote Read Buffer Current Address Register (MXVR_RRDB_CURR_ADDR)

MXVR Pattern Registers

The MXVR has two sets of Pattern Registers: Pattern 0 Registers (PR0) and Pattern 1 Registers (PR1). Each set of Pattern Registers contains a data register and an enable register. The pattern matching registers define a pattern which a synchronous DMA channel will search for in the incoming datastream when the DMA channel is in one of the Synchronous Packet modes. The MXVR_DMAx_CONFIG registers allow the “start pattern” to be defined by either PR0 or PR1 and the “stop pattern” to be defined by either PR0 or PR1. The patterns can be from one to four bytes long and can be enabled in a bit-wise manner to allow “don't cares”.

MXVR Pattern Data 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.

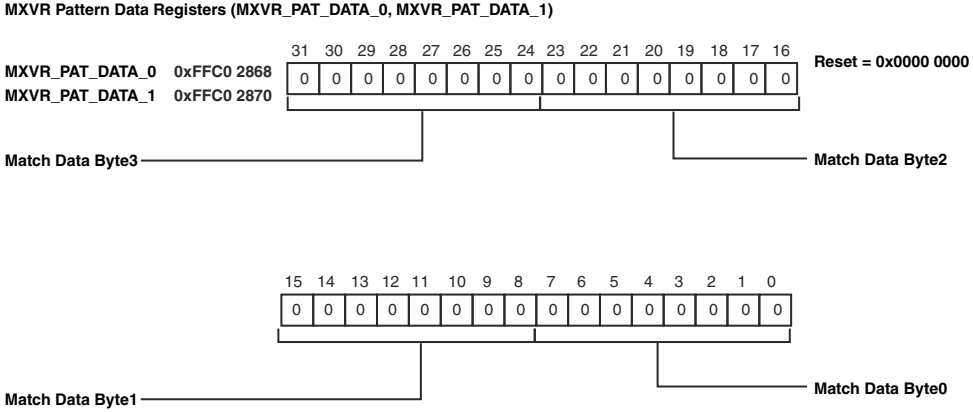


Figure 21-38. MXVR Pattern Data Registers (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1)

MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)

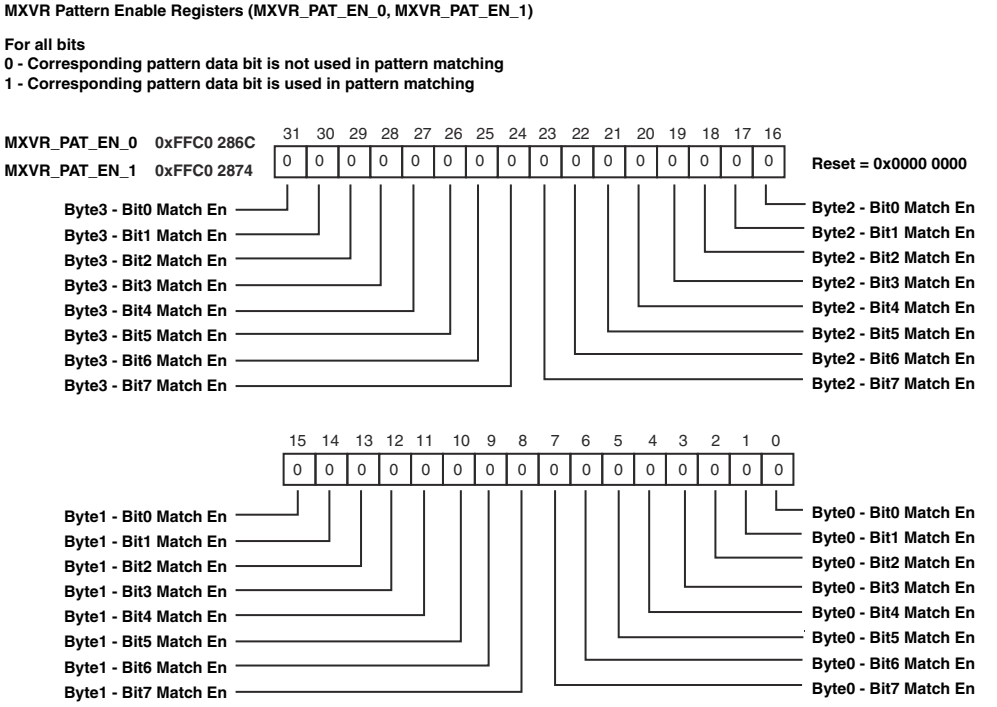


Figure 21-39. MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)

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. 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 has been started, the frame counter will pause until the MXVR is back in frame lock. The value written must be between 0x0001 and 0xFFFF. Once the frame counter decrements to zero, the corresponding Frame Counter Zero (FCZ0 or FCZ1) bit in the MXVR_INT_STAT_0 register will change to "1" and the Status Change Interrupt will assert if the corresponding Frame Counter Zero Interrupt Enable (FCZ0EN, or FCZ1EN) bit in the MXVR_INT_EN_0 register is set to "1". The FCZ0 and FCZ1 bits in the MXVR_INT_STAT_0 register are sticky bits which must be written

with a "1" in order to clear the bit and clear the interrupt. The frame counters can be stopped and reset at any time by writing 0x0000 to the MXVR_FRAME_CNT_x register and no interrupt will be generated. Reading the MXVR_FRAME_CNT_x will return the current value of the frame counter.

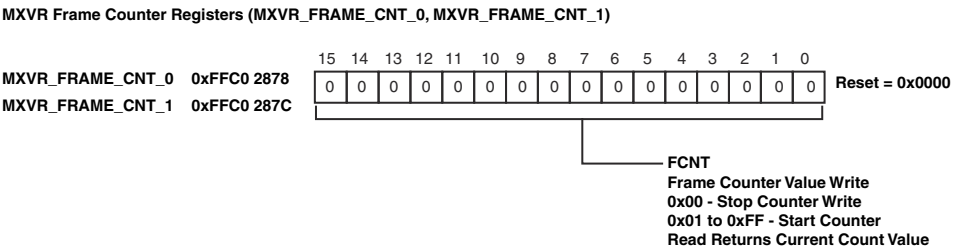


Figure 21-40. MXVR Frame Counter Registers (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1)

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 addition, the MXVR_ROUTING_x registers may be used to mute one or more transmitted physical channels. When a synchronous data channel is muted, the data transmitted on that channel will be 0x00.

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, MXVR_ROUTING_1 contains channels 4 to 7, and so on. Figure 21-41 on page 21-99 shows MXVR_ROUTING_0 as an example of the register format.

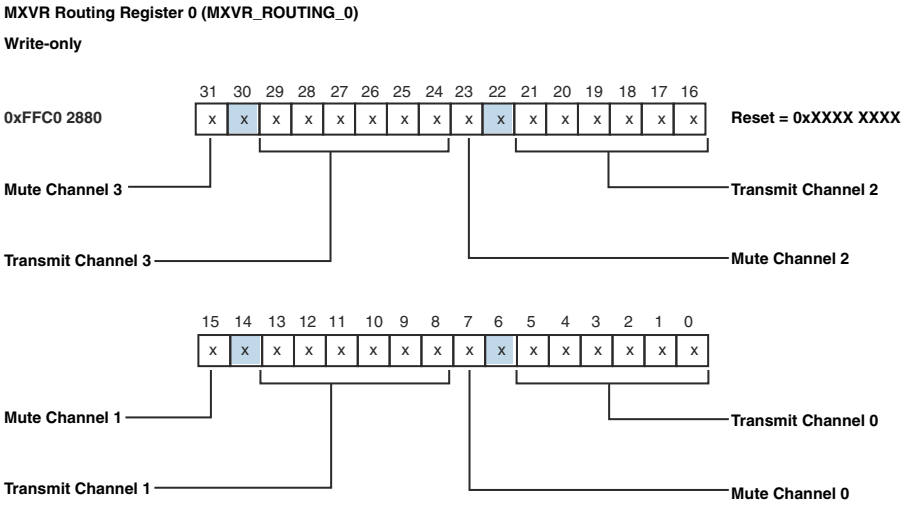


Figure 21-41. MXVR Routing Register 0 (MXVR_ROUTING_0)

The routing function can only be used for synchronous data channels (channel numbers less than 4 * RSB) when the MXVR is enabled and transmitting in Active Mode with a Synchronous Delay of two frames (when MXVREN = “1”, MTXEN = “1”, ACTIVE = “1” and SDELAY = "1" in the MXVR_CONFIG register). The muting function can be used for synchronous data channels when the MXVR is enabled and transmitting in Active Mode (MXVREN=”1”, MTXEN=”1”, and ACTIVE=”1”).

The MXVR_ROUTING_x registers 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 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 preambles are received when the Allocation Table is being distributed over the control message channel. Two block preambles out of every sixty-four block preambles are for Allocation Table distribution blocks.

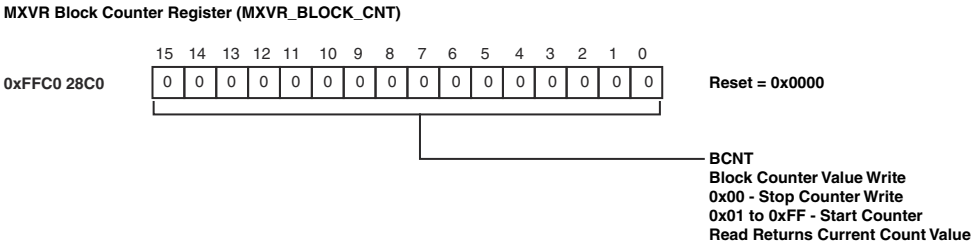


Figure 21-42. MXVR Block Counter Register (MXVR_BLOCK_CNT)

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 has been 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.

General Operation

The following sections describe MXVR general operations.

Network Services Software

Network Services Layer 1 and Layer 2 software has been developed for the MXVR on the ADSP-BF539 processor which meets the MOST Core Compliance specification. It is recommended that this software be used if the MXVR is to be operated in a MOST compliant network. Contact Analog Devices for more information on the Network Services software stack.

Network Activity Detection

Network activity detection is done to indicate whether a node is receiving an active datastream. Typically an ADSP-BF539 MXVR master node will be triggered to start up the network based on an external event (i.e. car ignition, power switch, etc.), while an ADSP-BF539 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 PHY Transmitter will be turned on, and the MXVR slave node will lock onto the incoming datastream. Once incoming network activity (circling the ring network) is detected by the master node, the MXVR master will lock onto the incoming datastream.

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 the ADSP-BF539 processor is powered-on that there is network activity. Note that the first two network activity detection methods can be utilized to generate interrupts even when the MXVR is disabled

The first method can be used in the case where the active-low status output of the PHY Receiver is connected to the MXVR $\overline{\text{MRXON}}$ input. When the PHY Receiver detects no network activity, the status output is set to "0" and when the PHY Receiver detects network activity, the status output is set to "1".

When the PHY Receiver 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-BF539 processor from Hibernate State if the MXVRWE bit in the VR_CTL register (see "Voltage Regulator Control Register" on page 8-27) has been set to "1". 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 PHY Receiver 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-BF539 processor to enter a low-power state.

In the second method for detecting network activity the MXVR detects edges on the MRX input. If a single rising or falling edge is detected on the MRX input, the MXVR will set the NACT bit in the MXVR_STATE_0 register to "1" indicating that there is network activity. If there are no rising or falling edges detected on the MRX input for 40 SCLK cycles, the MXVR will set the NACT bit to "0" indicating there is no network activity.

When the MXVR first detects network activity, the NACT bit will transition from low to high. The low to high transition of the NACT bit will set the NI2A bit in the MXVR_INT_STAT_0 register to "1" and if the NI2AEN bit in the MXVR_INT_EN_0 register is set to "1", an MXVR Status interrupt will be generated. The MXVR Status interrupt can also be programmed in the SIC_IWR1 to wake the core from the Idle state.

When the MXVR detects a cessation of network activity, the NACT bit will transition from high to low. The high to low transition of the NACT bit will set the NA2I bit in the MXVR_INT_STAT_0 register to "1" and if the NA2IEN bit in the MXVR_INT_EN_0 register is set to "1", an MXVR Status interrupt

will be generated. This interrupt on the cessation of network activity could be used to trigger the ADSP-BF539 processor to enter a low-power state.

The third method for network activity detection is handled completely outside of the ADSP-BF539 processor. In this method the PHY Receiver status output controls the power supply for the ADSP-BF539 processor. When the PHY Receiver status output indicates that there is no network activity, the power supply for the ADSP-BF539 processor is gated off. When the PHY Receiver status output indicates that there is network activity, the power supply for the ADSP-BF539 processor is turned on. Once the reset to the ADSP-BF539 processor negates after the power-on-reset, the software can assume that there is network activity.

Node Initialization

Prior to starting up the MXVR PLL and enabling the MXVR, the MXVR_CONFIG, MXVR_PLL_CTL_0, MXVR_PLL_CTL_1, MXVR_ROUTING_x, and the buffer start address registers must be initialized. The initialization of the MXVR_CONFIG register differs between a node to be started up in Master mode and a node to be started up in Slave mode. The initialization of the MXVR_PLL_CTL_0, MXVR_PLL_CTL_1, and the MXVR_ROUTING_x registers is the same for Master and Slave mode.

Master Mode Initialization of the MXVR_CONFIG Register

The MXVREN bit should remain "0" (keeping the MXVR disabled) until the MXVR PLL has been 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 PHY Transmitter 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 PHY Transmitter 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_PLL_CTL_0 Register

The MXTALCEN and MXTALFEN bits are both reset to "1" to allow a crystal connected between MXI and MXO to start-up immediately following the negation of reset. If either or both of these bits were set to "0" to save power, they must be set to "1" prior to starting up the MXVR PLL. If a crystal is used, enough time should be allowed for the crystal to start-up prior to enabling the MXVR PLL. The MXTALMUL bits should be set based on the frequency of the crystal or clock driven into MXI. The MPLLEN, MMCLKEN, MPLLRSTB, MBCLKEN, MPLLCDR, INVRX, and MFSEN bits should be set to "0". The state of the other bits in the MXVR_PLL_CTL_0 register do not matter until the MXVR PLL is started up.

Initialization of the MXVR_PLL_CTL_1 Register

The MSTO, MHOGGD, and MSHAPEREN bits should be set to "0" prior to starting up the MXVR PLL. The other bits in the MXVR_PLL_CTL_1 register control the PLL counter which can be used regardless of the whether the PLL has been started up or whether the MXVR has been enabled. The PLL counter

runs on the system clock and can be used to wait the periods of time required to start-up the crystal and the periods of time required in the PLL start-up sequence.

Initialization of the MXVR_ROUTING_x Registers

Unless specific rerouting of synchronous data between received and transmitted physical channels is desired once the MXVR is enabled and activated, the Transmit Channel x fields should be written to forward each received channel to the corresponding transmitted channel. In addition, unless specific channel muting is desired, the Mute Channel x fields should be programmed to disable muting. For example, the MXVR_ROUTING_x registers could be written to forward all channels and disable all muting as follows:

```
*pMXVR_ROUTING_0 = 0x0302 0100;

*pMXVR_ROUTING_1 = 0x0706 0504;

*pMXVR_ROUTING_2 = 0x0B0A 0908;

…

*pMXVR_ROUTING_13 = 0x3736 3534;

*pMXVR_ROUTING_14 = 0x3B3A 3938;
```

Initialization of the Buffer Start Address Registers

The control message transmit and receive buffers, the asynchronous packet transmit and receive buffers, and the remote read buffer should be allocated space in L1 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.

PLL Start-Up Sequence

Once the clock supplied to the MXVR either by a crystal connected between MXI and MXO or by an external oscillator driving MXI is stable and at frequency, the MXVR PLL can be started-up.

When the MXVR PLL is being started up after reset or when network activity is detected—the PLL which will be used for clock recovery must be put into frequency multiply mode to first lock at the frequency 1024 * Fs and then should be switched into clock recovery mode once the incoming datastream is being received.

To start up the MXVR PLL from a completely disabled state and frequency lock at 1024 * Fs, the following steps should be followed:

1. MPLLEN, MPLLRSTB, MSTO and MHOGGD should be set to "0" (MXVR PLL completely disabled). MPLLCDR should be set to "0" to select frequency multiply mode.

   If the MXVR will be operated as a Master, the PLLMS bit should be set to "1". Or if the MXVR will be operated as a Slave the PLLMS bit should be set to "0". The MXTALMUL bits should be set to b#11 and either a crystal should be connected between MXI and MXO or an oscillator connected to MXI supplying a 1024 * Fs clock and it should be at frequency.

2. MPLLEN should be set to "1" in the MXVR_PLL_CTL_0 register to turn on reference voltages and currents in the MXVR PLL.

3. Wait 1 µsec.

4. MPLLRSTB should be set to "1" in the MXVR_PLL_CTL_0 register to release the MXVR PLL from its reset state.

5. Wait 25 µsec.

6. MSTO should be set to "1" to enable the MXVR PLL voltage controlled oscillator.

7. Wait 8 ms.
8. PLL will be frequency locked at the frequency 1024 * F_s.

Once the MXVR PLL has been started up, the MXVR output clocks MMCLK, MBCLK, and MFS can programmed and enabled. When the MXVR is in either Master or Slave mode, the MXVR output clocks are created from the receive clock which is which is recovered from the incoming datastream by the MXVR PLL.

The following steps should be followed to program the MXVR output clocks once the MXVR PLL is frequency locked:

1. MMCLKEN, MBCLKEN, and MFSEN in the MXVR_PLL_CTL_0 register should all be set to "0" (toggling disabled on the MXVR output clocks).
2. Write the MMCLKMUL, MBCLKDIV, MFSDIV, MFSSEL, and MFSSYNC fields in the MXVR_PLL_CTL_0 register to define the frequency and relationship of the MXVR output clocks.
3. Wait 1 µsec.
4. Write to the MMCLKEN, MBCLKEN, and MFSEN bits in the MXVR_PLL_CTL_0 register to enable the individual MXVR output clocks to start toggling as desired. Set MMCLKEN to "1" to enable MMCLK. Set MBCLKEN to "1" to enable MBCLK. Set MFSEN to "1" to enable MFS.

Network Lock

The steps for setting up the MXVR PLL for clock recovery mode, enabling the MXVR, and attempting to lock onto the incoming datastream are slightly different for Master nodes and Slave nodes. The steps for Master node and the steps for Slave node are described below.

For a Master node:

Once the MXVR PLL is at frequency, the MXVR can be enabled, the MXVR transmit can be enabled, and the PHY Transmitter can be turned on. This is accomplished by writing to the MXVR_CONFIG register and setting the MXVREN bit to "1", the MTXEN bit to "1", and the MTXONB bit to "0". All other bits in the MXVR_CONFIG register should be left programmed as described in "Node Initialization" on page 21-105. Once the MXVR_CONFIG register has been written the MXVR will begin transmitting the network datastream on the MTX output pin and the PHY Transmitter will transmit the datastream to the first slave node in the network.

Once the MXVR master node detects incoming network activity (circling the ring network), the master node can put the MXVR PLL in clock recovery mode. The following steps should be followed to put the MXVR PLL in clock recovery mode:

1. MHOGGD should be set to "1" in the MXVR_PLL_CTL_1 register
2. MPLLCDR should be set to "1" in the MXVR_PLL_CTL_0 register

Once the master node has put the MXVR PLL in clock recovery mode, the MXVR will be attempting to lock onto the incoming datastream. The FLOCK, BLOCK, and SBLOCK bits in the MXVR_STATE_0 register indicate the current lock level of the MXVR. In addition state changes on the lock level state bits are indicated by the FU2L, FL2U, BU2L, BL2U, SBU2L, and SBL2U interrupt event bits in the MXVR_INT_STAT_0 register. These interrupt events can generate an MXVR Status interrupt if enabled.

For a Slave node:

Once the slave node has detected incoming network activity and the MXVR PLL is at frequency, the MXVR PLL can be put in clock recovery mode. The following steps should be followed to put the MXVR PLL in clock recovery mode:

1. MHOGGD should be set to "1" in the MXVR_PLL_CTL_1 register
2. MPLLCDR should be set to "1" in the MXVR_PLL_CTL_0 register

Once the MXVR PLL has been put in clock recovery mode, the MXVR can be enabled, the MXVR transmit can be enabled, and the PHY Transmitter can be turned on. This is accomplished by writing to the MXVR_CONFIG register and setting the MXVREN bit to "1", the MTXEN bit to "1", and the MTXONB bit to "0". All other bits in the MXVR_CONFIG register should be left programmed as described in "Node Initialization" on page 21-105. Once the MXVR_CONFIG register has been written the MXVR will begin forwarding the network datastream on the MTX output pin and the PHY Transmitter will transmit the datastream to the next slave node or back to the master node if it is the last slave node in the ring.

Once the MXVR PLL has been put in clock recovery mode and the MXVR has been enabled, the MXVR will be attempting to lock onto the incoming datastream. The FLOCK, BLOCK, and SBLOCK bits in the MXVR_STATE_0 register indicate the current lock level of the MXVR. In addition state changes on the lock level state bits are indicated by the FU2L, FL2U, BU2L, BL2U, SBU2L, and SBL2U interrupt event bits in the MXVR_INT_STAT_0 register. These interrupt events can generate an MXVR Status interrupt if enabled.

Network Initialization

Once the network has been locked, the Master node typically changes the value of the MSB in the MXVR_CONFIG register. Once the MSB field has been changed, the Master will distribute the new synchronous boundary over

the network. The update of the RSB value in each of the slave nodes and in the master node is used to indicate that the network lock is stable and the ring is closed. In the slave nodes, the update of the RSB value will cause the SBU interrupt event to be asserted. Note that once the network is in operation, a special procedure must be followed to dynamically change the synchronous boundary without disrupting the asynchronous packet channel.

The MXVR will automatically determine the node position, node delay, maximum node position, and maximum node delay from the network. Once the lock level of the MXVR is such that these values can be determined, the fields and valid bits in the MXVR_POSITION, MXVR_DELAY, MXVR_MAX_POSITION, and MXVR_MAX_DELAY registers will be updated. In addition, the PRU, DRU, MPRU, and MDRU interrupt events in the MXVR_INT_STAT_0 register will assert when these values become valid or change.

The MXVR will also automatically receive the Allocation Table distributed by the Master node in the MXVR_ALLOC_x registers. The Master node in the network distributes its Allocation Table once every 1024 frames. Once the distribution of the Allocation Table has completed, the ATU interrupt event will assert. The assertion of the ATU interrupt event only indicates that the Allocation Table distribution has been received and does not indicate whether the Allocation Table has changed since the last distribution. In the Master node, the ATU interrupt event also asserts when a Resource Allocate or Resource De-Allocate control message has caused the Allocation Table to be updated.

For the logical address to be used in the address comparison for received control messages and receive asynchronous packets, the LADDR field should be written and the LVALID bit should be set to "1" in the MXVR_LADDR register. Note that software must determine the uniqueness of the logical address.

For the group address to be used in the address comparison for received control messages, the GADDRL field should be written and the GVALID bit should be set to "1" in the MXVR_AADDR 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 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 has been 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 has been 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 has been defined, the DMA channel can be configured. The MXVR has 8 DMA channels dedicated for synchronous data transmission and reception. All 8 DMA channels have the same functionality and any number of them can

be used simultaneously. In order to configure DMA channel x for 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.

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 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.

Asynchronous Packet Transmission

The MXVR Asynchronous Packet Transmit Buffer (APTB) is an area of memory that is allocated to hold an asynchronous packet to be transmitted. The APTB must reside in L1 Memory and the starting address of the APTB is programmed in the MXVR_APTB_START_ADDR register. Enough memory should be allocated for the largest asynchronous packet to be transmitted. The largest allowed asynchronous packet data length is 1014 bytes and with 12 bytes for packet priority, addressing, and length fields, the APTB must be 1026 bytes.

Once the asynchronous packet to be transmitted has been written to the APTB, the STARTAP bit in the MXVR_AP_CTL register should be set to "1" to trigger the MXVR to begin arbitration and transmission of the asynchronous packet. At that point the MXVR will start DMA’ing the asynchronous packet from the APTB into the MXVR and will begin arbitrating for the asynchronous packet channel.

While the MXVR is still arbitrating for the asynchronous packet channel, the asynchronous packet transmission can be cancelled by setting the CANCELAP bit to "1" in the MXVR_AP_CTL register. Once the STARTAP bit has been 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 has been 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 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 21-6.

Table 21-6. 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)

Table 21-6. Asynchronous Packet Transmit Buffer Field Offsets

APTB 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 has been free for more than one frame, the node with the lowest `POSITION` will win arbitration. A node which has won arbitration is not allowed to arbitrate on the first free frame after it has transmitted. In the case when more than one node is arbitrating for the asynchronous packet channel after another node has just completed transmitting an asynchronous packet, the next downstream node which is arbitrating will win the arbitration.

The `AP Destination Address` should be programmed to be the logical address or alternate address of the node that will receive the asynchronous packet.

Software must calculate the AP Length field based on the length of the asynchronous packet data being transmitted. The AP Length field is a length in quadlets and the value includes 6 bytes for the AP Source Address (2 bytes) and AP CRC (4 bytes). The AP Length field can be calculated based on the length of the AP Data field (in bytes) using the following formula:

$$AP\ Length = ((Length(AP\ Data)) + 6) \div 4$$

The AP Source Address can be programmed to be any address representing the transmitting node. However, it is recommended that the logical address of the transmitting node be use.

The AP Data field contains the data to be transmitted in the asynchronous packet. The amount of data transmitted can be from 1 byte to 1014 bytes.

Asynchronous Packet Reception

The MXVR Asynchronous Packet Receive Buffer (APRB) is an area of memory that is allocated to hold received asynchronous packets. The APRB must reside in L1 Memory and the starting address of the APRB is programmed in the MXVR_APRB_START_ADDR register. Enough memory should be allocated for two 1024-byte asynchronous packets to be stored (2048 total bytes). The asynchronous packets are of variable length (ranging from 8 bytes to 1024 bytes) so the Length of Data field must be read to determine where the end of each asynchronous packet is located.

As asynchronous packets are received by the MXVR the packets will be DMA'd into the APRB in a sequential manner (wrapping from the end back to the start). For example, APRB Entry 0 will be filled first, then APRB Entry 1, and then APRB Entry 0, etc. As each message is received, the corresponding APRBEx bit in the MXVR_AP_CTL register will be set to "1" by the MXVR indicating that receive buffer entry number x is full. Once software has read the asynchronous packet, the APRBEx bit should be cleared by writing a "1" to the corresponding bit position indicating that receive buffer entry x is now empty.

If a new asynchronous packet is arriving and the next sequential entry is full, the Asynchronous Packet Receive Buffer Overflow (APROF) bit in the MXVR_INT_STAT_0 register will be set to "1" and can conditionally generate an interrupt. The incoming packet which caused the overflow will be lost.

The two APRB entries are stored as address offsets to the APRB start address programmed in MXVR_APRB_START_ADDR register. The address offsets for the two APRB Entries are given in Table 21-7.

Table 21-7. Asynchronous Packet Receive Buffer Entry Offsets

APRB Entry Offset	APRB Entry Number
MXVR_APRB_START_ADDR + 0x000	AP Receive Buffer Entry 0
MXVR_APRB_START_ADDR + 0x400	AP Receive Buffer Entry 1

Received asynchronous packets are DMA'd to the next sequential APRB entry in L1 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 21-8. 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 21-8. Asynchronous Packet Receive Buffer Entry Field Offsets

APRB Entry Address Offsets	Field Name
0x00	AP Destination Address (Upper Byte)
0x01	AP Destination Address (Lower Byte)
0x02	AP Length (in quadlets)
0x03	Reserved
0x04	AP Source Address (Upper Byte)
0x05	AP Source Address (Lower Byte)
0x06 to AP Data End Offset	AP Data

Control Message Transmission

The MXVR Control Message Transmit Buffer (CMTB) is an area of memory that is allocated to hold a control message to be transmitted. The CMTB must reside in L1 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 has been 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 has been 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 has been DMA'd back to the CMTB, the MXVR will set the CMTS bit in the MXVR_INT_STAT_0 register to “1” and an interrupt can be conditionally generated. Note that regardless of the actual response value (i.e. Transmission Status) received back, the MXVR will set the CMTS bit to "1".

The control message is said to be successfully cancelled if the CANCELCM bit is set to "1" prior to the MXVR winning the arbitration for the control message channel. If the control message is successfully cancelled, the MXVR will set the CMTC bit to in the MXVR_INT_STAT_0 register to “1” and an interrupt can conditionally be generated.

There are six types of control messages: Normal, Remote Read, Remote Write, Resource Allocate, Resource De-Allocate, and Remote GetSource. All six types of control message contain the following fields: CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM Transmission Status.

The CM Priority is used in the control message arbitration process. The CM Priority can range from 0x00 to 0x0F with 0x00 being the lowest priority and 0x0F being the highest priority. If more than one node is arbitrating for the control messages channel at the same time, the control message being sent with the highest CM Priority will win the arbitration. If the control messages being sent have the same CM Priority, the node which has won arbitration the least will win the arbitration. If the control messages have the same CM Priority and the nodes sending the control messages have won arbitration an equal amount, then the node with the lowest POSITION value will win the arbitration.

The CM Destination Address should be programmed to be the logical address, physical address, or group address of the node that will receive the control message. The byte order of the CM Destination Address is such that it can be written with a word write.

The CM Source Address can be programmed to be any address representing the transmitting node. However, it is recommended that the logical address of the transmitting node be use. The byte order of the CM Source Address is such that it can be written with a word write.

The CM Message Type field determines which type of control message is being sent. Table 21-9 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 21-9. 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 21-10 gives the meaning of the transmission status values received back when single cast addressing is used.

Table 21-10. Single cast Transmission Status Encodings

CM Transmission Status	Meaning of Transmission Status
0x0000	No Response
0x1010	Transmission Successful
0x1111	Not Supported
0x2020	CRC Error
0x2121	Receive Buffer Full

Table 21-11 on page 21-124 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'ed together).

Table 21-11. 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

Table 21-11. Group Cast/Broadcast Transmission Status Encodings (Cont'd)

CM Transmission Status	Meaning of Transmission Status
0x3030	CRC Error and Transmission Successful
	CRC Error and Transmission Successful and No Response
0x3131	Transmission Successful and Receive Buffer Full
	Not Supported and CRC Error
	Not Supported and Receive Buffer Full
	Transmission Successful and Receive Buffer Full and No Response
	Not Supported and CRC Error and No Response
	Not Supported and Receive Buffer Full and No Response
	Transmission Successful and Not Supported and CRC Error
	Transmission Successful and Not Supported and Receive Buffer Full
	Transmission Successful and Not Supported and CRC Error and No Response
	Transmission Successful and Not Supported and Receive Buffer Full and No Response

Normal Control Message Transmission

The normal control message is used to transmit data between nodes. The `CM Priority`, `CM Destination Address`, `CM Source Address`, `CM Message Type` (0x00), and `CM Data` fields should be written to the `CMTB` at the address offsets given in Table 21-12.

The `CM Data` field contains the data payload to be sent from the source to the destination. For normal control messages sent using single cast addressing all 17 bytes of the `CM Data` field may be used for data transmission. For normal control messages sent using group cast or broadcast addressing, only the first 16 bytes of the `CM Data` field should be used for data transmission. The 17th byte should be used as a unique message ID so that the destination nodes can ignore retries once they have successfully

received the normal control message. Note that software must handle the transmission of retries by retransmitting the same normal control message with the same message ID and checking the transmission status received back.

Once a normal control message has been 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 21-12.

Table 21-12. 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 if 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 21-13 on page 21-128.

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 has been written to the CMTB and the STARTCM bit is set to "1", the CM Priority, CM Destination Address, CM Source Address, CM Message Type and CM Read Address fields are DMA'ed from the CMTB to the MXVR. Once the MXVR wins arbitration the remote read control message is sent over the control message channel.

The data read from the RRDB and the transmission status from the destination node is received back by the MXVR and is DMA'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 21-13.

Table 21-13. 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 21-14 on page 21-130.

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 has been written to the CMTB and the STARTCM bit is set to "1", the CM Priority, CM Destination Address, CM Source Address, CM Message Type, CM Write Address, CM Write Length and CM Write Data fields are DMA'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 21-14.

Table 21-14. 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 wo 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 21-15 on page 21-132.

For resource allocate control messages, the CM Destination Address field should be restricted to either the logical address or the physical address of the Master node.

The CM Allocate Number Requested field contains the number of channels that the transmitting node is requesting to be allocated. The CM Allocate Number Requested should be in the range from 0x01 to 0x08. (indicating the number of channels being requested). If more than 8 channels are needed, more than one resource allocate control message should be sent to the Master.

Once a resource allocate control message has been 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 21-15.

Table 21-15. 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 21-16 on page 21-133 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 21-16. 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 has been processed and available to be allocated. If the resource allocate control message is sent to a Slave node, the CM Allocate Number Free response will be 0x00.

The CM Allocate Channel List field will contain 8 bytes representing physical channel numbers. If the CM Allocate Status response was “Allocation Successful”, then the first byte of the CM Allocate Channel List

will be the first of the channels that was allocated and will be the Connection Label. If n channels were requested to be allocated (`CM Allocate Requested` = n), then the first n bytes in the `CM Allocate Channel List` will be the actual channels allocated. For example, if 3 channels were requested to be allocated and the allocation was successful, then the channel numbers stored in the `CM Allocate Channel List` at address offsets 0x0C, 0x0D, and 0x0E are the channels that were allocated.

If the destination node successfully receives the resource allocate control message, the transmission status of "Transmission Successful" will be returned. If the destination node detects a CRC error in the resource allocate control message, the allocation will not take place (even if the CM Allocate Status response was "Allocation Successful") and the transmission status of "CRC Error" will be returned. If no node responds to the resource allocate control message, the transmission status of "No Response" will be returned.

Resource De-Allocate Control Message Transmission

The resource de-allocate control message is used to request dynamic de-allocation of synchronous channels from the Master node. A resource de-allocate control message can be sent to the Master to either de-allocate all the channels that are currently allocated or to de-allocate all the channels associated with a particular Connection Label. When a resource de-allocate control message is sent to the Master, the Master determines whether or not the request is valid, responds with the de-allocate status and updates the Allocation Table. All nodes in the network (including the Master itself), send resource de-allocate control messages to the Master to de-allocate channels. The `CM Priority`, `CM Destination Address`, `CM Source Address`, `CM Message Type` (0x04), and `CM De-Allocate Connection Label` fields should be written to the `CMTB` at the address offsets given in Table 21-17.

For resource de-allocate control messages, the CM Destination Address field should be restricted to either the logical address or the physical address of the Master node.

The CM De-Allocate Connection Label field contains either the Connection Label for the channels to be de-allocated or contains 0x7F if all channels are to be de-allocated. The CM De-Allocate Number Requested should be in the range from 0x00 to the uppermost synchronous channel number or can be 0x7F. The uppermost synchronous channel number can be determined by the following formula:

Uppermost Synchronous Channel Number = (4 * RSB) - 1

Once a resource de-allocate control message has been 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'ed 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'ed 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 21-17.

Table 21-17. Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x00	CM Priority
0x01	Reserved
0x02	CM Destination Address
0x04	CM Source Address
0x06	CM Message Type (Write 0x03)
0x07	Reserved (Write 0x00)

Table 21-17. Resource De-Allocate Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
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 21-18 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 21-18. CM De-Allocate Status Encodings

CM De-Allocate Status	Meaning of CM De-Allocate Status
0x01	De-Allocation Successful
0x02	Destination Busy

Table 21-18. CM De-Allocate Status Encodings

CM De-Allocate Status	Meaning of CM De-Allocate Status
0x04	De-Allocation Request Incorrect
0x05	Wrong Destination

If the destination node successfully receives the resource de-allocate control message, the transmission status of “Transmission Successful” will be returned. If the destination node detects a CRC error in the resource de-allocate control message, the de-allocation will not take place (even if the CM De-Allocate Status response was “De-Allocation Successful”) and the transmission status of “CRC Error” will be returned. If no node responds to the resource de-allocate control message, the transmission status of “No Response” will be returned.

Remote GetSource Control Message Transmission

The remote GetSource control message is used to determine which node is transmitting data on a particular physical channel. A remote GetSource control message can be sent using broadcast addressing and the node which is transmitting on the channel specified in the `CM GetSource Channel` field will respond with its physical address, logical address and group address. In addition, by setting the `CM GetSource Channel` field to 0xFF and sending a remote GetSource control message to a node using single cast addressing, the destination node will respond with its physical address, logical address, and group address. The `CM Priority`, `CM Destination Address`, `CM Source Address`, `CM Message Type` (0x04), and `CM GetSource Channel` fields should be written to the `CMTB` at the address offsets given in Table 21-19.

For remote getsource control messages, the CM Destination Address field should normally be sent using broadcast addressing; however, single cast addressing may be used to request a particular node to return its physical, logical and group addresses by sending 0xFF in the CM GetSource Channel field.

The CM GetSource Channel field contains a physical channel number. The CM GetSoruce Channel should be in the range from 0x00 to the uppermost synchronous channel number or can be 0xFF. The uppermost synchronous channel number can be determined by the following formula:

Uppermost Synchronous Channel Number = (4 * RSB) - 1

Once a remote GetSource control message has been written to the CMTB and the STARTCM bit is set to "1", the CM Priority, CM Destination Address, CM Source Address, CM Message Type, and CM GetSource Channel fields are DMA'ed from the CMTB to the MXVR. Once the MXVR wins arbitration the remote GetSource control message is sent over the control message channel. The response and transmission status from the destination node is received back by the MXVR and is DMA'ed back to the CMTB. The response will be stored in the CM GetSource Physical Address (Low), CM GetSource Group Address (Low), CM GetSource Logical Address (Low) and CM GetSource Logical Address (High) fields and the transmission status will be stored in the CM Transmission Status field of the CMTB at the address offsets given in Table 21-19.

Table 21-19. 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)

Table 21-19. Remote GetSource Control Message Transmit Buffer Entry Field Offsets

CMTB Address Offsets	Field Name
0x07	Reserved (Write 0x00)
0x08	CM GetSoruce 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

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

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

If the destination node responds to the remote getsource control message, the node will return the low byte of its group address (`GADDRL`) in the `CM GetSource Group Address` (Low) field.

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

If the remote getsource control message was sent and a response was received back and there was not a CRC error, the transmission status of "Transmission Successful" will be returned. If the remote getsource control message was sent and a response was received back but there was a CRC error, the transmission status of "CRC Error" will be returned. If the remote getsource control message was sent and no node responded, the transmission status of "No Response" will be returned. Note that since broadcast addressing is normally used when sending a remote getsource control message, it is possible that multiple nodes may respond and therefore, the transmitting node will receive back the response sent from the closest upstream node that responded.

Control Message Reception

The following sections describe the various types of control message reception.

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 Memory and the starting address of the CMRB is programmed in the MXVR_CMRB_START_ADDR register. Enough memory should be allocated for sixteen 24-byte messages to be stored (384 total bytes).

As normal control messages are received by the MXVR the normal control messages will be DMA'd into the CMRB in a sequential manner (wrapping from the end back to the start). For example, CMRBE0 will be filled first, then CMRBE1, ... , then CMRB15, then CMRE0, etc. As each message is received, the corresponding CMRBEx bit in the MXVR_CM_CTL register will be set to "1" by the MXVR indicating that receive buffer entry number x is full. Once

software has read the normal control message, the CMRBEx bit should be cleared by writing a "1" to the corresponding bit position indicating that receive buffer entry x is now empty.

If a new normal control message is arriving and the next sequential entry is full, the Control Message Receive Buffer Overflow (CMRBOF) bit in the MXVR_INT_STAT_0 register will be set to "1" and can conditionally generate an interrupt. The incoming message which caused the overflow will be lost and the Transmission Status will be returned to the transmitter indicating that the receive buffer was full. Note that an overflow will occur if the next sequential entry is full regardless of whether other entries in the CMRB are empty.

The 16 CMRB Entries are stored as address offsets to the CMRB start address programmed in the MXVR_CMRB_START_ADDR register. The address offsets for the 16 CMRB Entries are given in Table 21-20.

Table 21-20. Control Message Receive Buffer Entry Offsets

CMRB Entry Offset	CMRB Entry Number
MXVR_CMRB_START_ADDR + 0x000	CM Receive Buffer Entry 0
MXVR_CMRB_START_ADDR + 0x016	CM Receive Buffer Entry 1
MXVR_CMRB_START_ADDR + 0x02C	CM Receive Buffer Entry 2
MXVR_CMRB_START_ADDR + 0x16 * x	CM Receive Buffer Entry x
MXVR_CMRB_START_ADDR + 0x14A	CM Receive Buffer Entry 15

Received normal control messages are DMA'd to the next sequential CMRB Entry in L1 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 21-21.

Table 21-21. 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 memory that is allocated to allow other nodes to remotely read from and write to the ADSP-BF539 processor 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 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 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'd to the RRDB, the Remote Write Complete (RWRC) status bit will assert and an interrupt can be conditionally generated.

The start address of the RRDB is programmed in MXVR_RRDB_START_ADDR register. The received CM Write Data will be written into the RRDB Data field at the offset specified by the received CM Write Address. The received CM Write Address will be written into the RRDB Write Address field and the received CM Write Length will be written into the RRDB Write Length field so that when the remote write completes, software can easily determine which bytes have been remotely written. Table 21-22 gives the offsets of the RRDB Data, the RRDB Write Address, and the RRDB Write Length fields in the RRDB.

Table 21-22. 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 has been processed by the MXVR.

If a Resource Allocate control message is received by the MXVR when in Master mode, the ALIP bit in the MXVR_STATE_0 register will change to "1" to indicate a Resource Allocate control message is being processed. While the ALIP bit is a "1", the Allocation Table should not be read since the Allocation Table may be only partially updated. The MXVR will first determine whether the allocation request is correct, which channels are currently free in the Allocation Table, and whether there are enough channels available to satisfy the request. The MXVR will respond with the appropriate CM Allocate Status, CM Allocate Number Free, and CM Allocate Channel List. If no CRC error occurs during the Resource Allocate control message, the MXVR will update its Allocation Table to reflect the allocation request. Once the Allocation Table has been 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 has been 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 deallocation 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 deallocation request. Once the Allocation Table has been updated in the Master, the ATU bit in the MXVR_INT_STAT_0 register will change to "1". Note that the Master only distributes its Allocation Table to the Slave nodes once every 1024 frames.

If a Resource De-Allocate control message is received by the MXVR when in Slave mode, the MXVR will respond with the CM De-Allocate Status of “Wrong Destination”.

Remote GetSource Reception

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

If a Remote GetSource control message is received by the MXVR, the RGSIP bit in the MXVR_STATE_0 register will change to "1" to indicate a Remote GetSource control message is being processed. The MXVR will first determine whether it should respond to the Remote GetSource control message. The MXVR will respond if the MXVR is muting or routing data onto the channel specified in the CM GetSource Channel field or if the CM GetSource Channel field is 0xFF. The MXVR is muting channel n when the Channel Mute n bit in the appropriate MXVR_ROUTING_x register is set to "1". The MXVR is routing data onto channel n when the Transmit Channel n bit in the appropriate MXVR_ROUTING_x register is set to any value other than n. If the CM GetSource Channel field is 0xFF, the MXVR

will always respond regardless of what channels are being muted or routed. The MXVR will not respond if the CM GetSource Channel field has a value between 4 * RSB and 0xFE.

When the MXVR responds to a Remote GetSource control message, the MXVR returns the low byte of its Physical Address, the low byte of its Group Address, and the high and low bytes of its Logical Address. Note that values in the POSITION, GADDRL, and LADDR fields are returned in the response regardless of whether the corresponding valid bits are set to "1". Note that the Remote GetSource control message is normally sent as a broadcast message, so it is possible that more than one node could respond with one response overwriting another.

MXVR Low Power Operation

The ADSP-BF539 processor provides a number of mechanisms for dynamically controlling performance and power dissipation. The main mechanisms are controlling the voltage level of the processor through the on-chip voltage regulator, controlling the core clock and system clock frequencies, and controlling the operating mode of the core and the system PLL. See Chapter 8, "Dynamic Power Management" Within the ADSP-BF539 processor dynamic power management framework, the MXVR has six general power/functionality states as shown in Table 21-23.

Table 21-23. ADSP-BF539 Power/MXVR Functionality States

ADSP-BF539 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	No	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, MRXON, CANRX	Not Active
Power Gated Off to ADSP-BF539	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 memory.

These power/functionality states are listed in order from least power savings (Full On Mode) to greatest power savings (Power Gated Off). The functionality of the MXVR in each of these states is described in the following sections.

Full On Mode

When the ADSP-BF539 processor is operated in the Full On mode, the MXVR is fully functional and can be operated in any of its modes. While in the Full On mode, the system PLL generates the core clock and system clock. For power savings the core clock frequency can be reduced based on the minimum core processing performance required by the application and the system clock frequency can be reduced based on the minimum internal and external bus bandwidth required by the application. Once the minimum core clock frequency and system clock frequency required for the application is known, the voltage level of the on-chip voltage regulator may be lowered to further reduce power consumption. The tables giving

the minimum operating voltage level for a given core clock/system clock frequency combination can be found in the *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

The MXVR utilizes its DMA channels to transfer data to and from L1 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 memory bandwidth used in the system (i.e. memory DMA operations to/from L1, other peripheral DMA to/from L1, and core accesses to/from L1). 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 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 PLL and the MXVR Crystal Oscillator or `MXI` clock input can be disabled to reduce power consumption. To disable the MXVR PLL the `MPLLRSTB` bit should be set to 0 and the `MPLLEN` bit should be set to 0 in the `MXVR_PLL_CTL_0` 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_PLL_CTL_0` 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_PLL_CTL_0` register to gate off the `MXI` clock in the pad.

Active Mode

When the ADSP-BF539 processor is operated in the Active Mode, the MXVR is fully functional and can be operated in any of its modes. While in the Active Mode the system PLL is bypassed and the core clock and system clock run at the frequency of CLKIN. The core clock frequency must be operated at a high enough frequency to support the core processing performance required by the application and the system clock frequency must be operated at a high enough frequency to support the internal and external bus bandwidth required by the application. Based on the core clock frequency and system clock frequency, the voltage level of the on-chip voltage regulator may be lowered to further reduce power consumption. The tables giving the minimum operating voltage level for a given core clock/system clock frequency combination can be found in the *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet*.

The MXVR utilizes its DMA channels to transfer data to and from L1 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 memory bandwidth used in the system (i.e. memory DMA operations to/from L1, other peripheral DMA to/from L1, and core accesses to/from L1). 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 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 PLL and the MXVR Crystal Oscillator or MXI clock input can be disabled to reduce

power consumption. To disable the MXVR PLL the MPLLRSTB bit should be set to 0 and the MPLLEN bit should be set to 0 in the MXVR_PLL_CTL_0 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_PLL_CTL_0 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_PLL_CTL_0 register to gate off the MXI clock in the pad.

Sleep Mode

When the ADSP-BF539 processor is operated in Sleep Mode, the MXVR may be operated in any of its modes, however it cannot transmit synchronous data, asynchronous packets, or control messages, and it cannot receive synchronous data, asynchronous packets, normal control messages, remote read control messages or remote write control messages. 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 deallocation system control messages and other background network functions.

Before the idle instruction is executed that causes the ADSP-BF539 processor to enter Sleep Mode, all MXVR synchronous data DMA channels should be disabled by writing the DMAENx bits to 0, asynchronous packet reception should be disabled by setting the APRXEN bit to 0, normal control message reception should be disabled by setting the NCMRXEN bit to 0, and remote write control message reception should be disabled by setting the RWRRXEN bit to 0.

In addition since DMA accesses to L1 are not allowed during Sleep Mode, the remote read DMA channel must also be disabled by programming the MXVR_RRDB_START_ADDR to a non-existent L1 address (such as 0xFF80 8000). This will cause a DERR event to be generated if a remote read control message is received which will keep the remote read DMA channel from accessing L1 memory. In this case, the MXVR will respond

to the remote read control message sending out whatever data was previously in the MXVR control message transmit FIFO. If there is a desire to have the MXVR respond to remote read control messages with a particular data pattern while the ADSP-BF539 processor is in Sleep Mode, a normal control message with the data payload holding some known data pattern can be started and then immediately cancelled (write the MXVR_CM_CTL register with STARTCM set to 1 and then do a second write to the MXVR_CM_CTL register with CANCELCM set to 1) prior to entering Sleep Mode.

Once the ADSP-BF539 processor has entered Sleep Mode, it can be woken up back into either the Full On mode or Active Mode by any system interrupt. The wake-up interrupt should be enabled within the peripheral and within the 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)
- PLL Counter, Frame Counter, or Block Counter time-outs (PCZ, FCZx, BCZ)
- Detection of network lock changes (SBU2L, SBL2U, BU2L, BL2U, FU2L, FL2U)
- Detection of network status changes (PRU, MPRU, DRU, MDRU, SBU, ATU)

Some examples of other interrupt sources that may typically be used to wake-up from Sleep Mode are:

- Real Time Clock events
- Timer time-out
- PFx pin edge or level
- Peripheral data reception or transmission

If the MXVR is going to be operated in All Bypass-MXVR Disabled mode while in Sleep Mode, the MXVR PLL and the MXVR Crystal Oscillator or MXI clock input can be disabled to reduce power consumption. To disable the MXVR PLL the MPLLRSTB bit should be set to 0 and the MPLLEN bit should be set to 0 in the MXVR_PLL_CTL_0 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_PLL_CTL_0 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_PLL_CTL_0 register to gate off the MXI clock in the pad.

Deep Sleep Mode

When the ADSP-BF539 processor is operated in Deep Sleep Mode, the core clock and the system clock are disabled, therefore, the MXVR may only be operated in All Bypass-MXVR Disabled mode. The MXVR is by default in the All Bypass-MXVR Disabled mode after reset, or the All Bypass-MXVR Disabled mode may be entered by writing the MXVREN bit to 0. Within the All Bypass-MXVR Disabled mode, the MRX input is directly connected to the MTX output, so the MOST network can still be active while an ADSP-BF539 processor slave node is in Deep Sleep Mode.

Deep Sleep Mode can be exited only by a Real Time Clock interrupt or hardware reset. A Real Time Clock interrupt causes the ADSP-BF539 processor to transition to the Active Mode and the core will continue

executing the code following the idle instruction. A hardware reset will cause the ADSP-BF539 processor to exit Deep Sleep Mode and begin the hardware reset booting sequence.

If a crystal is connected between MXI and MXO, the MXVR Crystal Oscillator may also be disabled to eliminate the power consumption of the crystal oscillator during Deep Sleep Mode. This is accomplished by setting the MXTALFEN and MXTALCEN bits to 0 in the MXVR_PLL_CTL_0 register before executing the idle instruction that causes the ADSP-BF539 processor to enter Deep Sleep Mode. The reset sequence when exiting from deep sleep mode will cause the MXTALFEN and MXTALCEN bits to be reset to 1, so the MXVR Crystal Oscillator will start up during the reset sequence. If an external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_PLL_CTL_0 register to gate off the MXI clock in the pad.

Hibernate State

When the ADSP-BF539 processor is operated in Hibernate State, the on-chip voltage regulator is turned off and the internal power supplies (VDDINT, MPIVDD) transition to 0V. The only power that is used in this mode is the leakage current on the external power supplies (VDDEXT, MXEVDD) 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-BF539 processor is in Hibernate State.

There are four wake-up sources that can wake the ADSP-BF539 processor from Hibernate State:

- Real Time Clock Interrupt
- Asserting Hardware Reset

- Asserting the $\overline{\text{MRXON}}$ input low (typically connected to the MOST FOR Status output)
- Asserting the CANRX input low

The 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-BF539 processor to enter Hibernate State. In the VR_CTL register the WAKE bit should be set to 1 to allow wake-up on Real Time Clock interrupts, the MXVRWE bit should be set to 1 to allow wake-up on the assertion of $\overline{\text{MRXON}}$, and the CANWE bit should be set to 1 to allow wake-up on the assertion of CANRX. When any one of the enabled wake-up source events occurs, the on-chip voltage regulator will turn on and the ADSP-BF539 processor will begin the hardware reset booting sequence. A hardware reset will always take the processor out of Hibernate State.

If a crystal is connected between MXI and MXO, the MXVR Crystal Oscillator may also be disabled to eliminate the power consumption of the crystal oscillator during Hibernate State. This is accomplished by setting the MXTALFEN and MXTALCEN bits in the MXVR_PLL_CTL_0 register to 0 before executing the idle instruction that causes the ADSP-BF539 processor to enter Hibernate State. The reset sequence when exiting from Hibernate State will cause the MXTALFEN and MXTALCEN bits to be reset to 1, so the MXVR Crystal Oscillator will start up during the reset sequence. If an external oscillator is used to supply the MXI clock, the external oscillator should be disabled or the MXTALCEN bit should be set to 0 in the MXVR_PLL_CTL_0 register to gate off the MXI clock in the pad.

Power Gating the ADSP-BF539 Processor

To achieve the lowest possible power consumption for a MOST node, the external power supplies (VDDEXT, MXEVDD, RTCVDD) to the ADSP-BF539 processor should be gated off and pulled to 0V. This effectively reduces the ADSP-BF539 processor power consumption to zero. Typically the MOST FOR status output would be used to gate the ADSP-BF539 processor power supplies on and off based on the reception of modulated light.

22 SYSTEM DESIGN

This chapter provides hardware, software and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for pin information, including pin numbers for the 316-ball CSP_BGA package.

Recommendations for Unused Pins

Refer to *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for detailed pin descriptions.

Resetting the Processor

In addition to the Hardware Reset mode provided via the $\overline{\text{RESET}}$ pin, the processor supports several software reset modes. For detailed information on the various modes, see "System Reset and Power-up" on page 3-12.

The processor state after reset is described in "Reset State" on page 3-11.

Booting the Processor

The processor can be booted via a variety of methods. These include executing from external 16-bit memory, booting from a ROM configured to load code from 8-bit flash memory, or booting from a serial ROM (8-bit, 16-bit, or 24-bit address range). For more information on boot modes, see "Booting Methods" on page 3-18.

Figure 22-1 and Figure 22-2 show the connections necessary for 8-bit and 16-bit booting, respectively. Notice that the address connections are made in the same manner for both 8- and 16-bit peripherals. Only the lower byte of each 16-bit word is accessed if byte-wide memory is used.

For example, on core reads of the form:

```
R0 = W[P0] (Z) ;   /*P0 points to a 16-bit aligned ASYNC memory
location*/
```

only the lower 8 bits of `R0` contain the actual value read from the 8-bit device.

For core writes of the form:

```
W[P0] = R0.L ;   /*P0 points to a 16-bit aligned ASYNC memory
location*/
```

The 8-bit value to be written to the 8-bit device should be first loaded into the lower byte of `R0`.

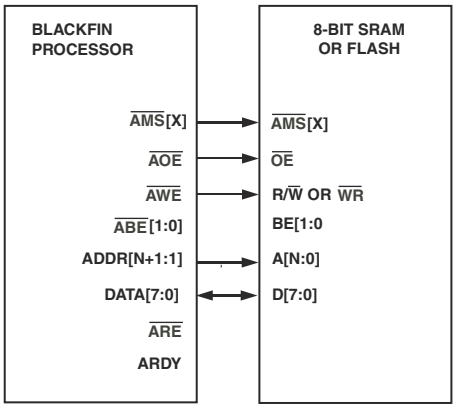


Figure 22-1. Interface to 8-bit SRAM or Flash

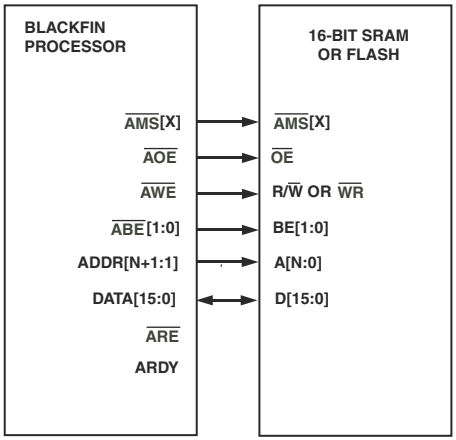


Figure 22-2. Interface to 16-bit SRAM or Flash

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's `CLKIN` pin. It is not possible to halt, change, or

operate CLKIN below the specified frequency during normal operation. The processor uses the clock input (CLKIN) to generate on-chip clocks. These include the core clock (CCLK) and the peripheral clock (SCLK).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 8, "Dynamic Power Management".

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the System interrupt Assignment registers (SIC_IARx). For more information, see "System Interrupt Assignment (SIC_IARx) Registers" on page 4-33.

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, see "Nesting of Interrupts" on page 4-55.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particu-

lar resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the Test and Set Byte (Atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 22-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 22-1. Query Semaphore

```
/* Query semaphore. Denotes "Busy" if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread,
and all other contending threads are postponed because semaphore
value at [P0] is nonzero. Current thread could write thread_id to
semaphore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an
operating system call to reschedule the current thread. */
```

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to Chapter 7, “Chip Bus Hierarchy”.

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to Chapter 7, “Chip Bus Hierarchy”.

External Memory Design Issues

This section describes design issues related to external memory.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit wide SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 22-3 shows the system interconnect required to support 16-bit memories. The programming model must ensure that data is only accessed on 16-bit boundaries.

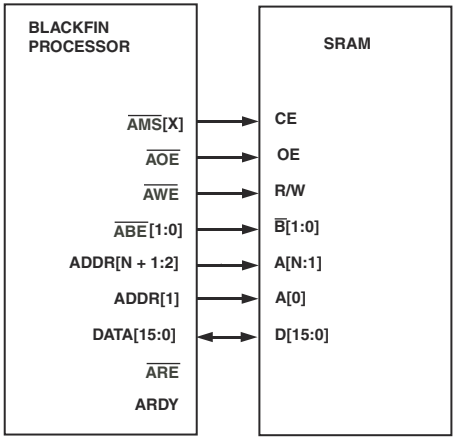


Figure 22-3. Interface to 16-bit SRAM

Using SDRAMs Smaller than 16M byte

It is possible to use SDRAMs smaller than 16M byte on the ADSP-BF539 processor, as long as it is understood how the resulting memory map is altered. Figure 22-4 shows an example where a 2M byte SDRAM (512K x

16 bits x 2 banks) is mapped to the external memory interface. In this example, there are 11 row addresses and 8 column addresses per bank. Referring to Table , the lowest available bank size (16M byte) for a device with 8 column addresses has 2 Bank Address lines (`IA[23:22]`) and 13 Row Address lines (`IA[21:9]`). Therefore, 1 processor bank Address line and 2 Row Address lines are unused when hooking up to the SDRAM in the example. This causes aliasing in the processor's external memory map, which results in the SDRAM being mapped into non-contiguous regions of the processor's memory space.

Referring to the table in Figure 22-4, note that each line in the table corresponds to 2^{19} bytes, or 512K byte. Thus, the mapping of the 2M byte SDRAM is non-contiguous in Blackfin memory, as shown by the memory mapping in the left side of the figure.

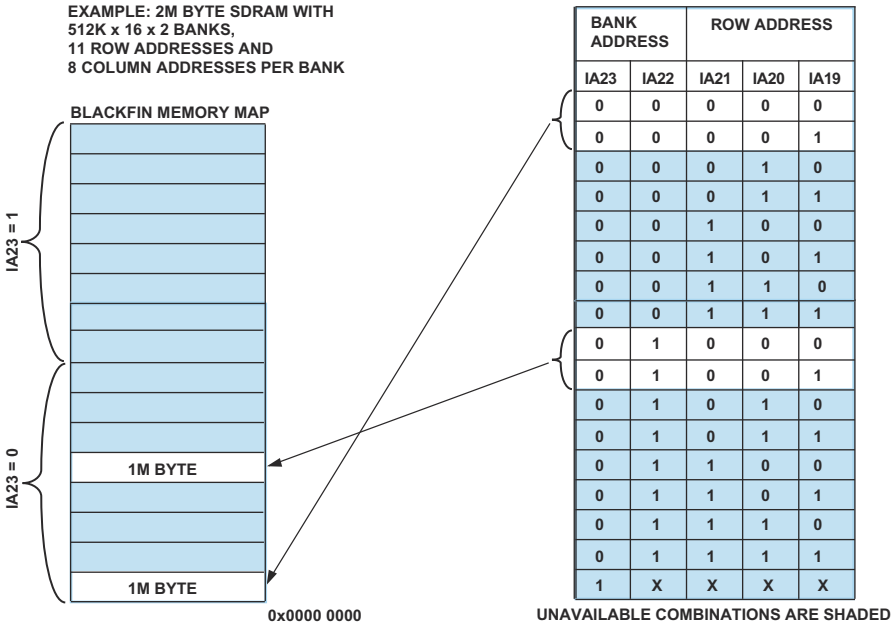


Figure 22-4. Using Small SDRAMs

Managing SDRAM Refresh During PLL Transitions

Since the processor's SDRAM refresh rate is based on the SCLK frequency, lowering SCLK after configuring SDRAM can result in an improper refresh rate, which could compromise the data stored in SDRAM. Raising SCLK after configuring SDRAM, however, would merely result in a less efficient use of SDRAM, since the processor would just refresh the memory at an unnecessarily fast rate.

In systems where SDRAM is used, the recommended procedure for changing the PLL VCO frequency is:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to Self-Refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired PLL programming sequence (refer to Chapter 8, "Dynamic Power Management" for details).
4. After the wakeup occurs that signifies the PLL has settled to the new VCO frequency, reprogram the SDRAM Refresh Rate control register (EBIU_SDRRC) with a value appropriate to the new SCLK frequency.
5. Bring the SDRAM out of Self-Refresh mode by clearing the SRFS bit of EBIU_SDGCTL. If it is desired to change the SDRAM Mode register, write these changes to EBIU_SDGCTL as well, making sure the PSSE bit is set.

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

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

4. Reprogram the SDRAM Refresh Rate control register (EBIU_SDRRC) with a value appropriate to the new SCLK frequency.
5. Bring the SDRAM out of Self-Refresh mode by clearing the SRFS bit of EBIU_SDGCTL. If it is desired to change the SDRAM Mode register, write these changes to EBIU_SDGCTL as well, making sure the PSSE bit is set.

Note steps 2 and 4 are not strictly necessary if changing SCLK to a higher value, but they should always be performed when decreasing SCLK.

For more information on SDRAM refresh, refer to "SDRAM Controller (SDC)" in Chapter 18, External Bus Interface Unit.

Avoiding Bus Contention

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

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

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

High Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Point-to-Point Connections on Serial Ports

Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and may require source termination for longer distances.

You can add a series termination resistor near the pin for point-to-point connections. Typically, serial port applications use this termination method when distances are greater than 6 inches. For details, see the reference source in "Recommended Reading" on page 22-15 for suggestions on transmission line termination. Also, see *ADSP-BF539/ADSP-BF539F Blackfin Embedded Processor Data Sheet* for rise and fall time data for the output drivers.

Signal Integrity

The capacitive loading on high-speed signals should be reduced as much as possible. Loading of buses can be reduced by using a buffer for devices that operate with wait states (for example, DRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes.

Signal run length (inductance) should also be minimized to reduce ringing. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the Printed Circuit Board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes. These planes should be in the center of the PCB.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Design for lower transmission line impedances to reduce crosstalk and to allow better control of impedance and delay.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the V_{DDEXT} and V_{DDINT} pins of the package as shown in Table 22-5 on page 22-14. Use. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane inside the package footprint of the processor (under-

neath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

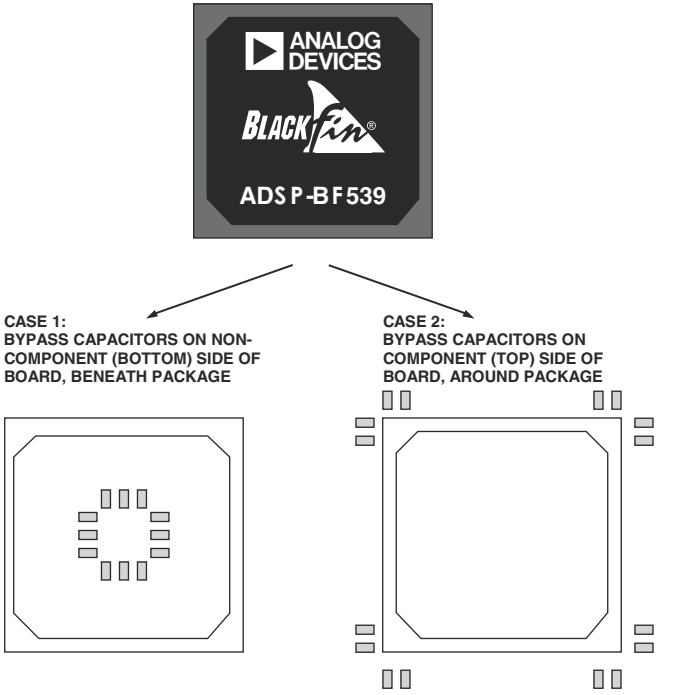


Figure 22-5. Bypass Capacitor Placement

Connect the power plane to the power supply pins directly with minimum trace length. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced. In addition, there should be several large tantalum capacitors on the board.

Designs can use either bypass placement case or combinations of the two. Designs should try to minimize signal feedthroughs that perforate the ground plane.

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent source of information and practical ideas. Topics covered in the book include:

- High-speed Properties of Logic Gates
- Measurement Techniques
- Transmission Lines
- Ground Planes and Layer Stacking
- Terminations
- Vias
- Power Systems
- Connectors

- Ribbon Cables
- Clock Distribution
- Clock Oscillators

23 BLACKFIN PROCESSOR DEBUG

The Blackfin processor's debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in Table 23-1.

Table 23-1. Blackfin Debug Features

Debug Feature	Description
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Trace History	Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.

In addition, information that the Watchpoint Unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The Watchpoint Unit contains these memory-mapped registers (MMRs), which are accessible in Supervisor and Emulator modes:

- The Watchpoint status register (WPSTAT)
- Six Instruction Watchpoint Address registers (WPIA[5:0])
- Six Instruction Watchpoint Address Count registers (WPIACNT[5:0])
- The Instruction Watchpoint Address control register (WPIACTL)
- Two Data Watchpoint Address registers (WPDA[1:0])
- Two Data Watchpoint Address Count registers (WPDACNT[1:0])
- The Data Watchpoint Address control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Instruction Watchpoint Address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the Instruction Watchpoint Address Count registers, WPIACNT[5:0], are decremented on each match.

The six Instruction Watchpoint Address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.

The address ranges stored in WPIA0, WPIA1, WPIA2, WPIA3, WPIA4, and WPIA5 must satisfy these conditions:

- WPIA0 <= WPIA1
- WPIA2 <= WPIA3
- WPIA4 <= WPIA5

Two operations implement data watchpoints:

- The values in the two Data Watchpoint Address registers, WPDA[1:0], are compared to the address on the data buses.
- Corresponding count values in the Data Watchpoint Address Count registers, WPDACNT[1:0], are decremented on each match.

The two Data Watchpoint Address registers may be further grouped together into one data-address-range Watchpoint, WPDA[1:0].

The instruction and data count value registers must be loaded with the number of times the Watchpoint must match minus one. After the count value reaches zero, the subsequent Watchpoint match results in an exception or emulation event.

Note count values must be reinitialized after the event has occurred.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPAND bit in the WPIACTL register is set, then an event is triggered only when both an instruction address Watchpoint matches *and* a data address Watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or Watchpoint ranges match.

To enable the Watchpoint Unit, the WPPWR bit in the WPIACTL register must be set. If WPPWR = 1, then the individual watchpoints and Watchpoint ranges may be enabled using the specific enable bits in the WPIACTL and WPDACTL MMRs. If WPPWR = 0, then all Watchpoint activity is disabled.

Instruction Watchpoints

Each instruction Watchpoint is controlled by three bits in the WPIACTL register, as shown in Table .

Table 23-2. WPIACTL Control Bits

Bit Name	Description
EMUSWx	Determines whether an instruction-address match causes either an emulation event or an exception event.
WPICNTENx	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAENx	Enables the address Watchpoint activity.

When two watchpoints are associated to form a range, two additional bits are used, as shown in Table .

Table 23-3. WPIACTL Watchpoint Range Control Bits

Bit Name	Description
WPIRENxy	Indicates the two watchpoints that are to be associated to form a range.
WPIRINVxy	Determines whether an event is caused by an address within the range identified or outside of the range identified.

Code patching allows software to replace sections of existing code with new code. The Watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the WPIAn registers and setting the corresponding EMUSWx bit to trigger an exception. In the exception service routine, the WPSTAT register is read to determine which Watchpoint triggered the exception. Next, the code writes the start address of the new code in the RETX register, and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the WPSTAT MMR clears all the sticky status bits. The data value written is ignored.

WPIAn Registers

When the Watchpoint Unit is enabled, the values in the Instruction Watchpoint Address registers (WPIAn) are compared to the address on the instruction bus. Corresponding count values in the Instruction Watchpoint Address Count registers (WPIACNTn) are decremented on each match.

Figure 23-1 shows the Instruction Watchpoint Address registers, WPIA[5:0].

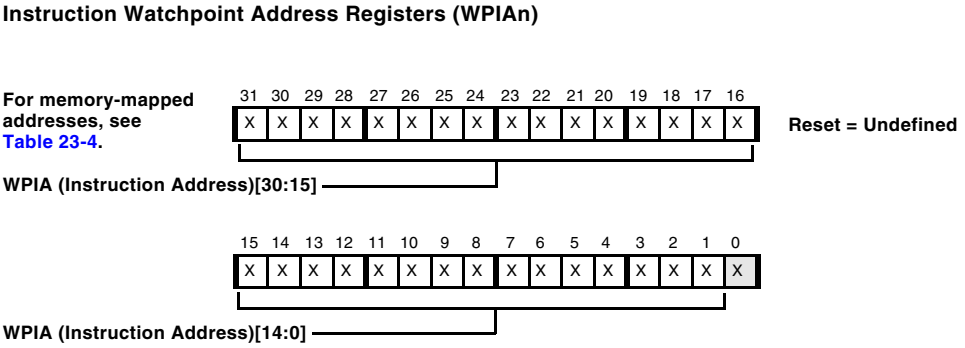


Figure 23-1. Instruction Watchpoint Address Registers

Table 23-4. Instruction Watchpoint register Memory-mapped Addresses

Register Name	Memory-mapped Address
WPIA0	0xFFE0 7040
WPIA1	0xFFE0 7044
WPIA2	0xFFE0 7048
WPIA3	0xFFE0 704C
WPIA4	0xFFE0 7050
WPIA5	0xFFE0 7054

WPIACNTn Registers

When the Watchpoint Unit is enabled, the count values in the Instruction Watchpoint Address Count registers (WPIACNT[5:0]) are decremented each time the address or the address bus matches a value in the WPIAn registers. Load the WPIACNTn register with a value that is one less than the number of times the Watchpoint must match before triggering an event (see Figure 23-2). The WPIACNTn register will decrement to 0x0000 when the programmed count expires.

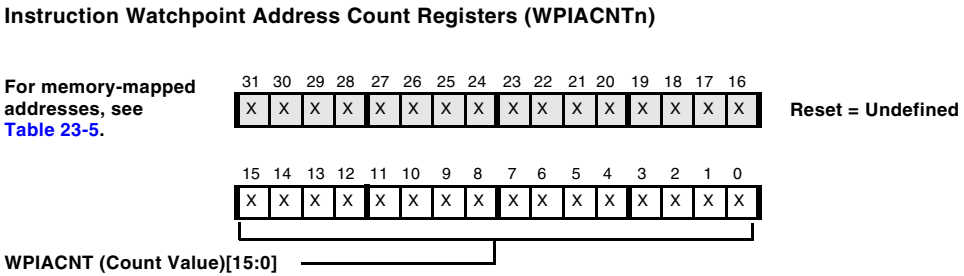


Figure 23-2. Instruction Watchpoint Address Count Registers

Table 23-5. Instruction Watchpoint Address Count register memory-mapped Addresses

Register Name	Memory-mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

WPIACTL Register

Three bits in the Instruction Watchpoint Address control register (WPIACTL) control each instruction Watchpoint. Figure 23-3 describes the upper half of the register. Figure 23-4 on page -9 describes the lower half of the register. For more information about the bits in this register, see "Instruction Watchpoints" on page 23-4.

The bits in the WPIACTL register have no effect unless the WPPWR bit is set.

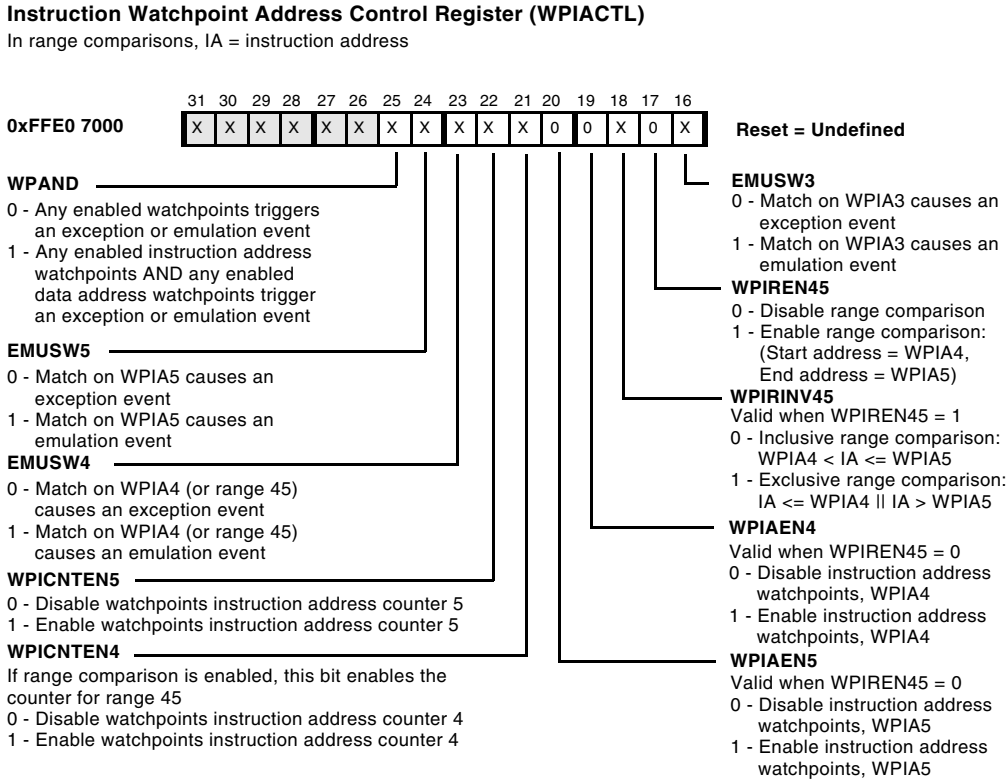


Figure 23-3. Instruction Watchpoint Address Control Register (WPIACTL)[31:16]

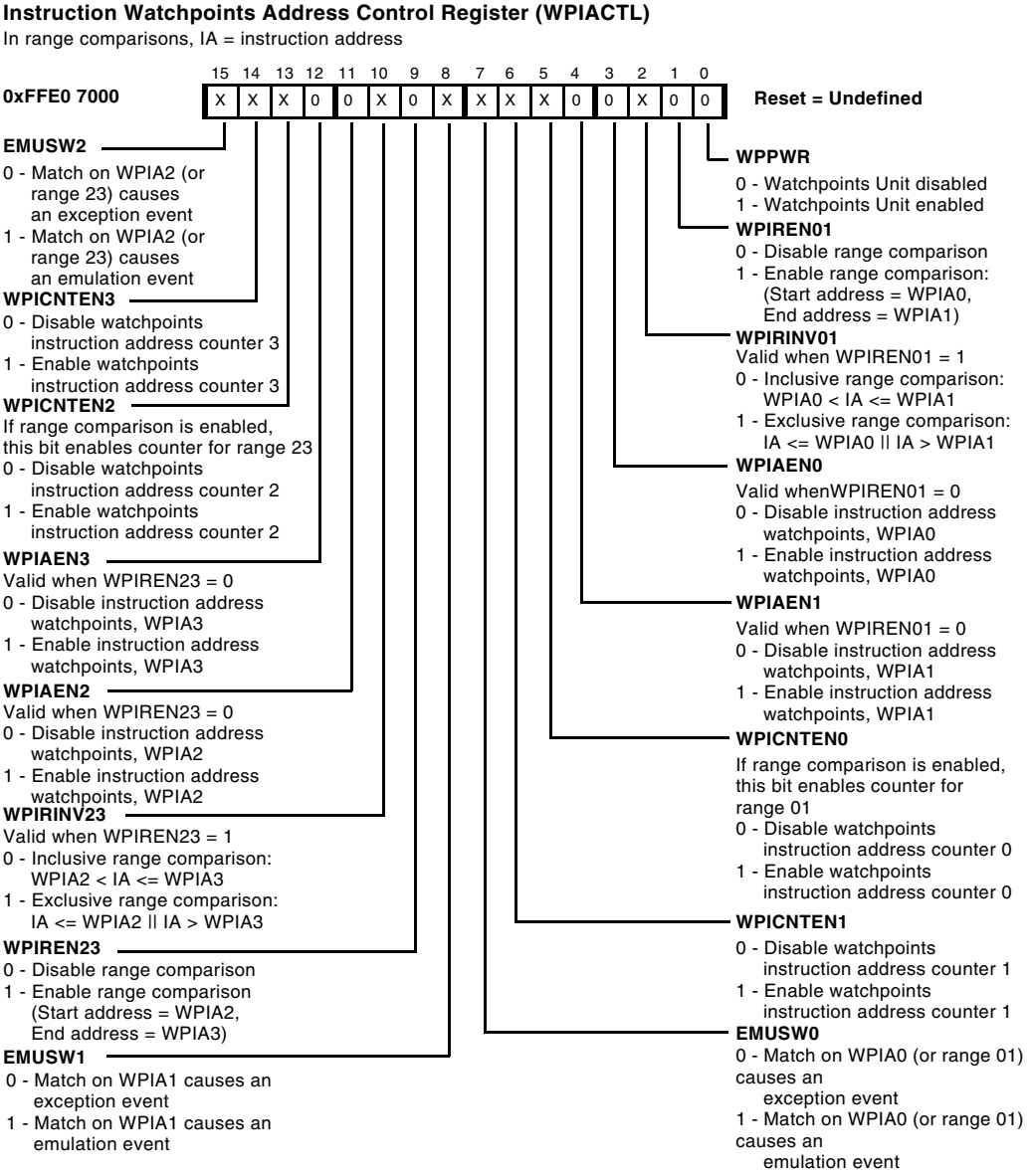


Figure 23-4. Instruction Watchpoint Address Control Register (WPIACTL)[15:0]

Data Address Watchpoints

Each data Watchpoint is controlled by four bits in the WPDACTL register, as shown in Table .

Table 23-6. Data Address Watchpoints

Bit Name	Description
WPDACCn	Determines whether the match should be on a read or write access.
WPDSRCn	Determines which DAG the unit should monitor.
WPDCNTENn	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAENn	Enables the data Watchpoint activity.

Alternatively, the watchpoint unit can be configured to monitor a range of data addresses. To enable this function, the WPDAEN0 and WPDAEN1 bits are not used and must be set to 0. Instead, the WPDREN01 and WPDRINV01 bits are used to configure the watchpoint unit, as described in Table 23-7.

Table 23-7. WPDACTL Watchpoint Control Bits

Bit Name	Description
WPDREN01	Indicates the two watchpoints associated to form a range.
WPDRINV01	Determines whether an event is caused by an address within the range identified or outside the range.

Note data address Watchpoints always trigger emulation events.

WPDAn Registers

When the Watchpoint Unit is enabled, the values in the Data Watchpoint Address registers (WPDAn) are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (WPDACNTn) are decremented on each match.

Figure 23-5 shows the Data Watchpoint Address registers, WPDA[1:0].

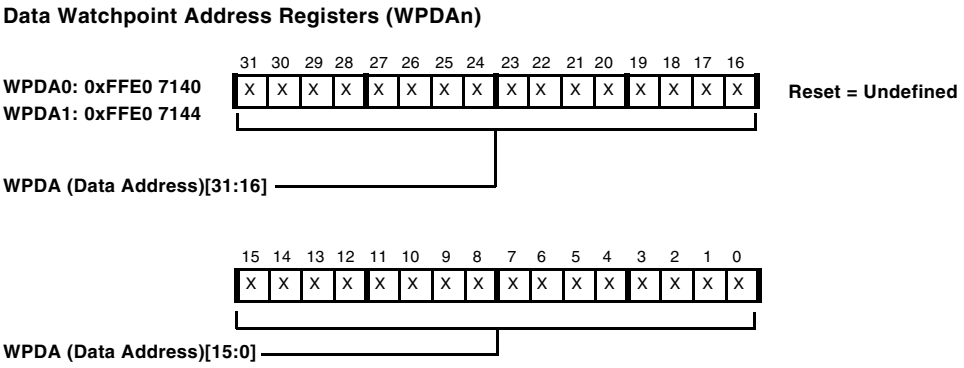


Figure 23-5. Data Watchpoint Address Registers

WPDACNTn Registers

When the Watchpoint Unit is enabled, the count values in the Data Watchpoint Address Count Value registers (WPDACNTn) are decremented each time the address or the address bus matches a value in the WPDAn registers. Load this WPDACNTn register with a value that is one less than the number of times the Watchpoint must match before triggering an event. The WPDACNTn register will decrement to 0x0000 when the programmed count expires. Figure 23-6 shows the Data Watchpoint Address Count Value registers, WPDACNT[1:0].

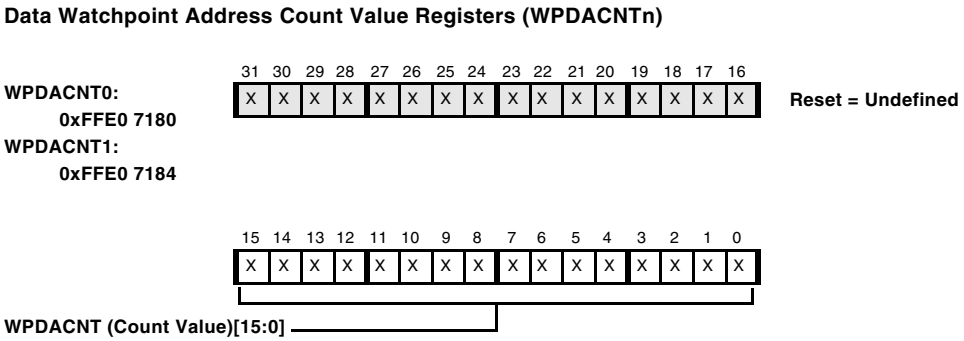


Figure 23-6. Data Watchpoint Address Count Value Registers

WPDACTL Register

For more information about the bits in the Data Watchpoint Address control register (WPDACTL), see "Data Address Watchpoints" on page 23-10.

WPSTAT Register

The Watchpoint status register (WPSTAT) monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a Watchpoint or Watchpoint range matches, this register reflects the source of the Watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

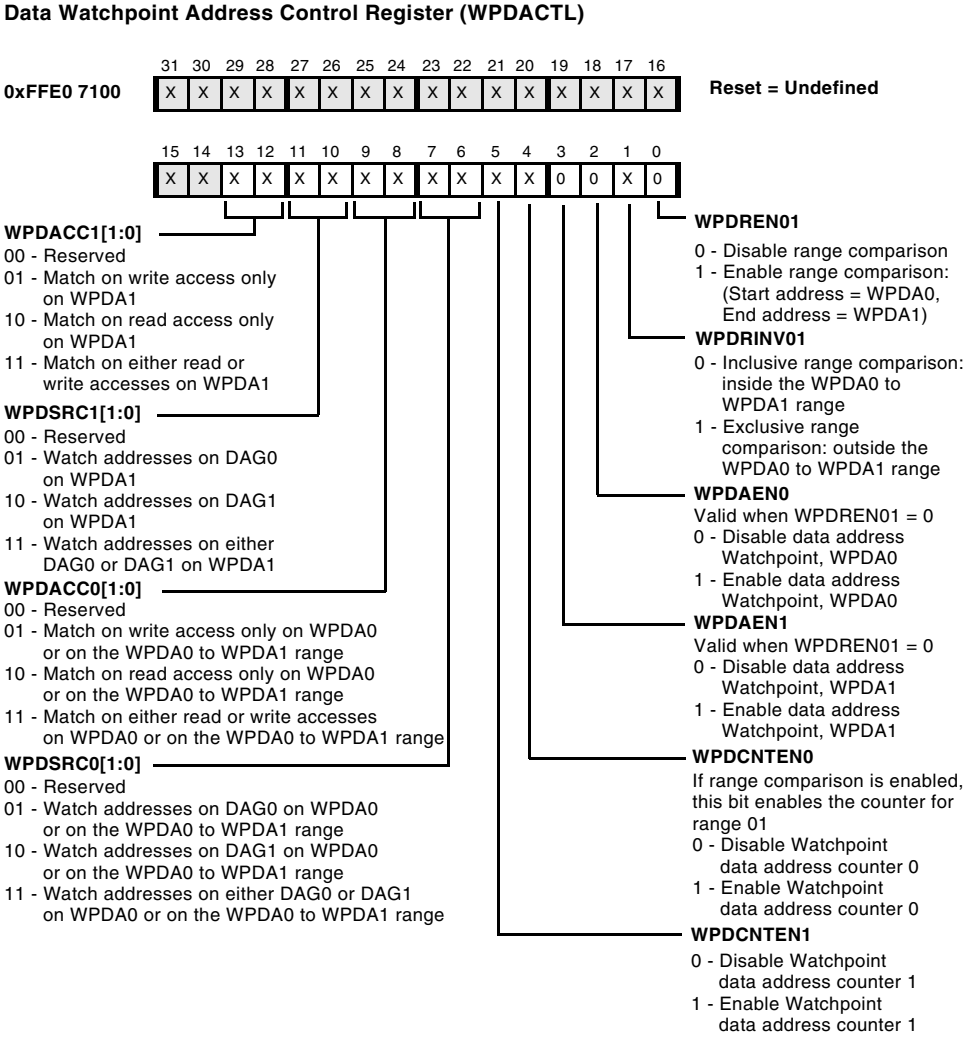


Figure 23-7. Data Watchpoint Address Control Register

Figure 23-8 shows the Watchpoint status register.

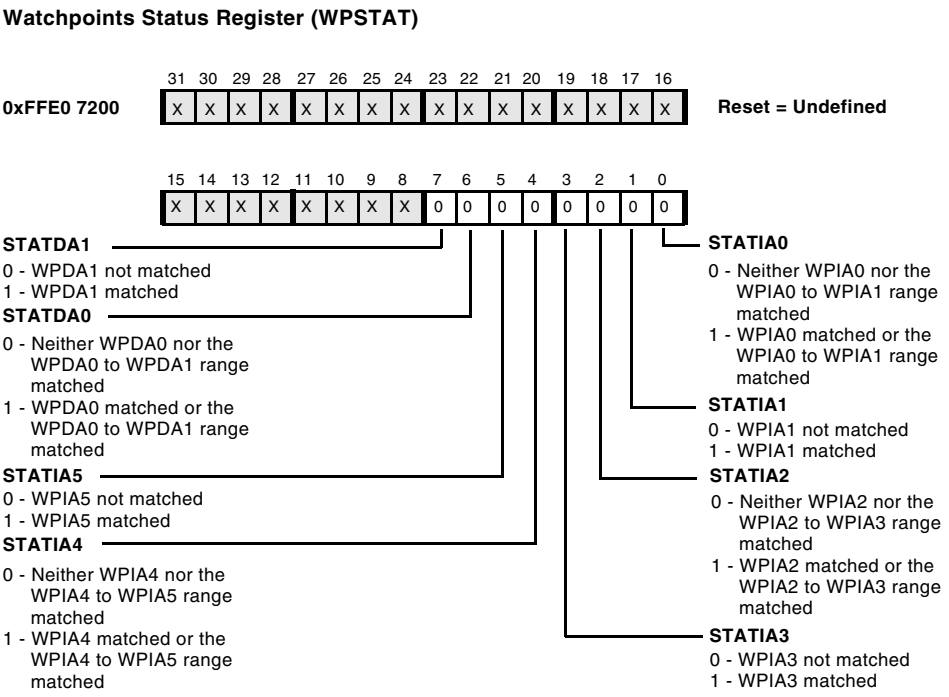


Figure 23-8. Watchpoint Status Register

Trace Unit

The Trace Unit stores a history of the last 16 changes in program flow taken by the program sequencer. The history allows the user to recreate the program sequencer's recent path.

The trace buffer can be enabled to cause an exception when full. The exception service routine associated with the exception saves trace buffer entries to memory. Thus, the complete path of the program sequencer since the trace buffer was enabled can be recreated.

Changes in program flow because of zero-overhead loops are not stored in the trace buffer. For debugging code that is halted within a zero-overhead loop, the iteration count is available in the Loop Count registers, LC0 and LC1.

The trace buffer can be configured to omit the recording of changes in program flow that match either the last entry or one of the last two entries. Omitting one of these entries from the record prevents the trace buffer from overflowing because of loops in the program. Because zero-overhead loops are not recorded in the trace buffer, this feature can be used to prevent trace overflow from loops that are nested four deep.

When read, the Trace Buffer register (TBUF) returns the top value from the Trace Unit stack, which contains as many as 16 entries. Each entry contains a pair of branch source and branch target addresses. A read of TBUF returns the newest entry first, starting with the branch destination. The next read provides the branch source address.

The number of valid entries in TBUF is held in the TBUFCNT field of the TBUFSTAT register. On every second read, TBUFCNT is decremented. Because each entry corresponds to two pieces of data, a total of 2 x TBUFCNT reads empties the TBUF register.

Discontinuities that are the same as either of the last two entries in the trace buffer are not recorded.

Because reading the trace buffer is a destructive operation, it is recommended that TBUF be read in a non-interruptible section of code.

Note, if single-level compression has occurred, the least significant bit (LSB) of the branch target address is set. If two-level compression has occurred, the LSB of the branch source address is set.

TBUFCTL Register

The Trace Unit is enabled by two control bits in the Trace Buffer control register (TBUFCTL) register. First, the Trace Unit must be activated by setting the TBUFPWR bit. If TBUFPWR = 1, then setting TBUFEN to 1 enables the Trace Unit.

Figure 23-9 describes the Trace Buffer control register (TBUFCTL). If TBUFOVF = 1, then the Trace Unit does not record discontinuities in the exception, NMI, and reset routines.

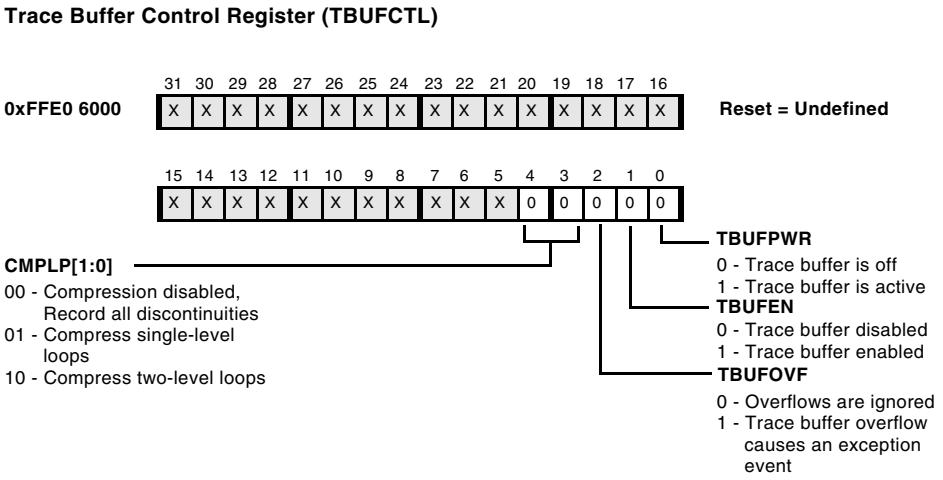


Figure 23-9. Trace Buffer Control Register

TBUFSTAT Register

Figure 23-10 shows the Trace Buffer status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by one.

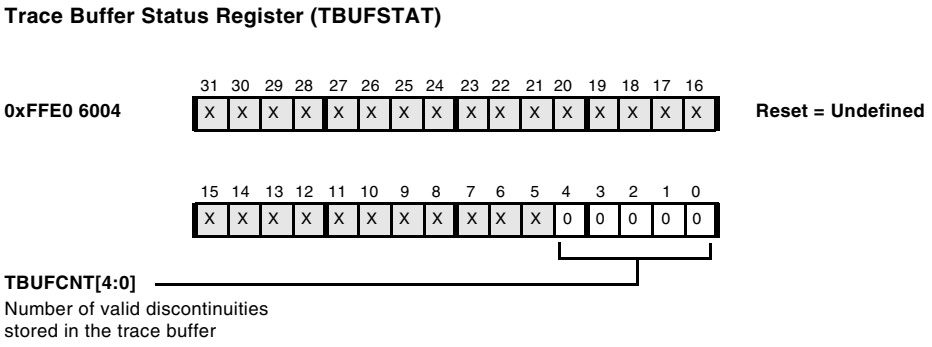


Figure 23-10. Trace Buffer Status Register

TBUF Register

Figure 23-11 shows the Trace Buffer register (TBUF). The first read returns the latest branch target address. The second read returns the latest branch source address.

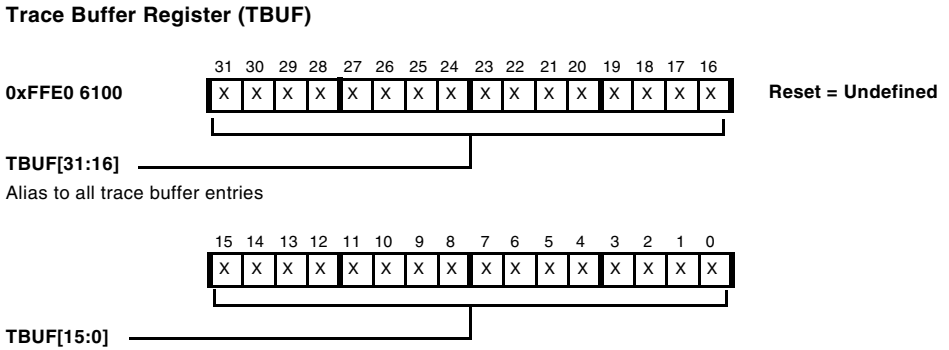


Figure 23-11. Trace Buffer Register

The Trace Unit does not record changes in program flow in:

- Emulator mode
- The exception or higher priority service routines (if TBUFOVF = 1)

  In the exception service routine, the program flow discontinuities may be read from TBUF and stored in memory by the code shown in Listing 23-1.

While TBUF is being read, be sure to disable the trace buffer from recording new discontinuities.

Code to Recreate the Execution Trace in Memory

Listing 23-1 provides code that recreates the entire execution trace in memory.

Listing 23-1. Recreating the Execution Trace in Memory

```
[--sp] = (r7:7, p5:2); /* save registers used in this routine */
p5 = 32;  /* 32 reads are needed to empty TBUF */
p2.l = buf;  /* pointer to the header (first location) of the
software trace buffer */
p2.h = buf;  /* the header stores the first available empty buf
location for subsequent trace dumps */

p4 = [p2++];  /* get the first available empty buf location from
the buf header */
p3.l = TBUF & 0xffff;  /* low 16 bits of TBUF */
p3.h = TBUF >> 16;  /* high 16 bits of TBUF */

lsetup(loop1_start, loop1_end) lc0 = p5;
loop1_start: r7 = [p3];  /* read from TBUF */
loop1_end: [p4++] = r7;  /* write to memory and increment  */
[p2] = p4;  /* pointer to the next available buf location is
saved in the header of buf */
(r7:7, p5:3) = [sp++];  /* restore saved registers */
```

Performance Monitoring Unit

Two 32-bit counters, the Performance Monitor Counter registers (PFCNTR[1:0]) and the Performance control register (PFCTL), count the number of occurrences of an event from within a processor core unit during a performance monitoring period. These registers provide feedback indicating the measure of load balancing between the various resources on the chip so that expected and actual usage can be compared and analyzed. In addition, events such as mispredictions and hold cycles can also be monitored.

PFCNTRn Registers

Figure 23-12 shows the Performance Monitor Counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1.

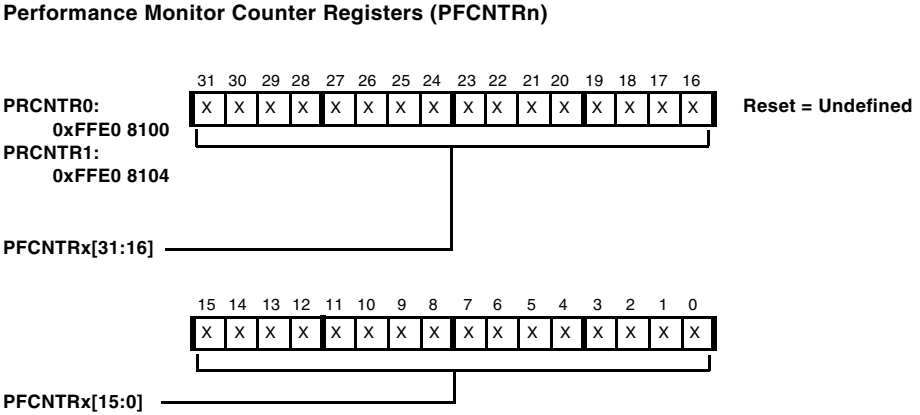


Figure 23-12. Performance Monitor Counter Registers

PFCTL Register

To enable the Performance Monitoring Unit, set the PFPWR bit in the Performance Monitor control register (PFCTL), shown in Figure 23-13. Once the unit is enabled, individual count-enable bits (PFCENn) take effect. Use the PFCENx bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the PEMUSWx bits to select the type of event triggered.

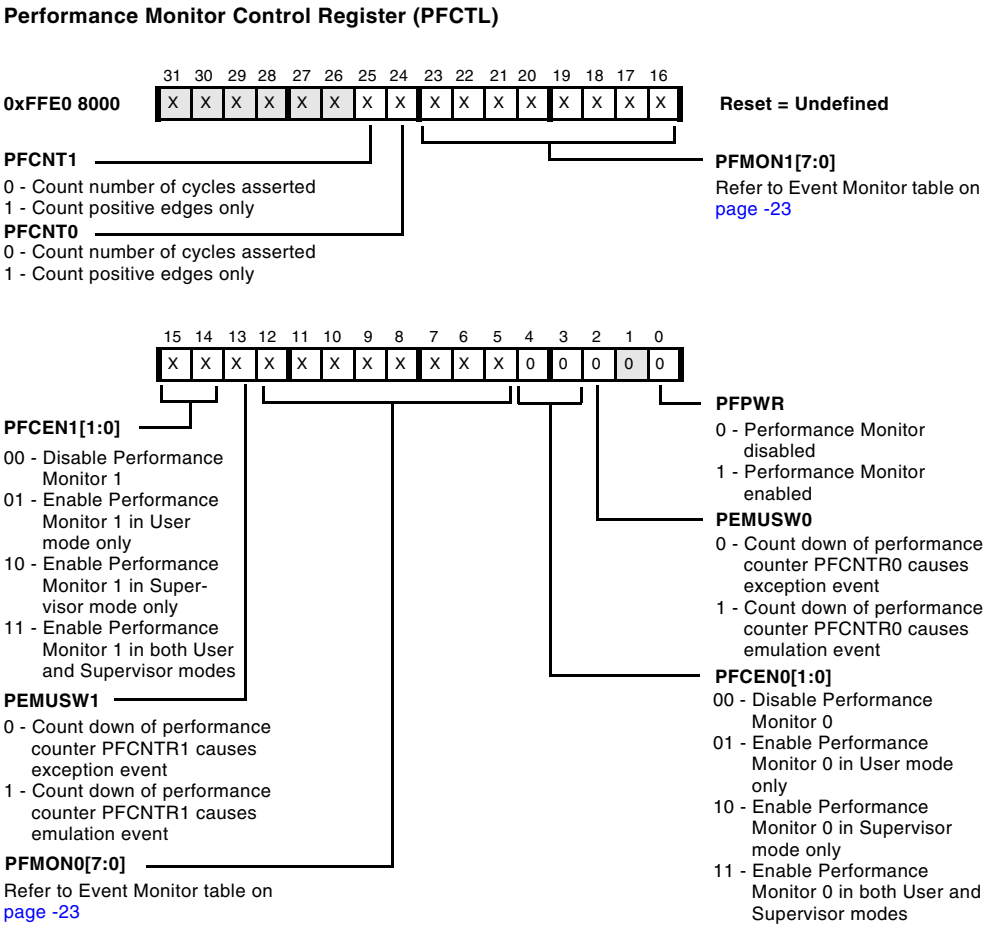


Figure 23-13. Performance Monitor Control Register

Event Monitor Table

Table identifies events that cause the Performance Monitor Counter registers (PFMON0 or PFMON1) to increment.

Table 23-8. Event Monitor Table

PFMONx Fields	Events That Cause the Count Value to Increment
0x00	Loop 0 iterations
0x01	Loop 1 iterations
0x02	Loop buffer 0 not optimized
0x03	Loop buffer 1 not optimized
0x04	PC invariant branches (requires trace buffer to be enabled, see "TBUFCTL Register" on page 23-16)
0x06	Conditional branches
0x09	Total branches including calls, returns, branches, but not interrupts (requires trace buffer to be enabled, see "TBUFCTL Register" on page 23-16)
0x0A	Stalls due to CSYNC, SSYNC
0x0B	EXCPT instructions
0x0C	CSYNC, SSYNC instructions
0x0D	Committed instructions
0x0E	Interrupts taken
0x0F	Misaligned address violation exceptions
0x10	Stall cycles due to read after write hazards on DAG registers
0x13	Stall cycles due to RAW data hazards in computes
0x80	Code memory fetches postponed due to DMA collisions (minimum count of two per event)
0x81	Code memory TAG stalls (cache misses, or FlushI operations, count of 3 per FlushI). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.

Table 23-8. Event Monitor Table (Cont'd)

PFMONx Fields	Events That Cause the Count Value to Increment
0x82	Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.
0x83	Code memory 64-bit words delivered to processor instruction assembly unit
0x90	Processor stalls to memory
0x91	Data memory stalls to processor not hidden by processor stall
0x92	Data memory store buffer full stalls
0x93	Data memory write buffer full stalls due to high-to-low priority code transition
0x94	Data memory store buffer forward stalls due to lack of committed data from processor
0x95	Data memory fill buffer stalls
0x96	Data memory array or TAG collision stalls (DAG to DAG, or DMA to DAG)
0x97	Data memory array collision stalls (DAG to DAG or DMA to DAG)
0x98	Data memory stalls
0x99	Data memory stalls sent to processor
0x9A	Data memory cache fills completed to bank A
0x9B	Data memory cache fills completed to bank B
0x9C	Data memory cache victims delivered from bank A
0x9D	Data memory cache victims delivered from bank B
0x9E	Data memory cache high priority fills requested
0x9F	Data memory cache low priority fills requested

Cycle Counter

The cycle counter counts CCLK cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.

The cycle counter is 64 bits and increments every cycle. The count value is stored in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits (LSBs) are stored in CYCLES. The most significant 32 bits (MSBs) are stored in CYCLES2.

To ensure read coherency, a read of CYCLES stores the current CYCLES2 value in a shadow register, and a subsequent read of CYCLES2 comes from the shadow register.

To enable the cycle counters, set the CCEN bit in the SYSCFG register. The following example shows how to use the cycle counter to benchmark a piece of code:

```
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;
/* Insert code to be benchmarked here. */
R2 = SYSCFG;
BITCLR(R2,1);
SYSCFG = R2;
```

CYCLES and CYCLES2 Registers

The Execution Cycle Count registers (CYCLES and CYCLES2) are shown in Figure 23-14. This 64-bit counter increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter's 64-bit count value. The most significant 32 bits are contained by CYCLES2.

The CYCLES and CYCLES2 registers are not system MMRs, but are instead system registers. See page -8 for a full listing of system registers.

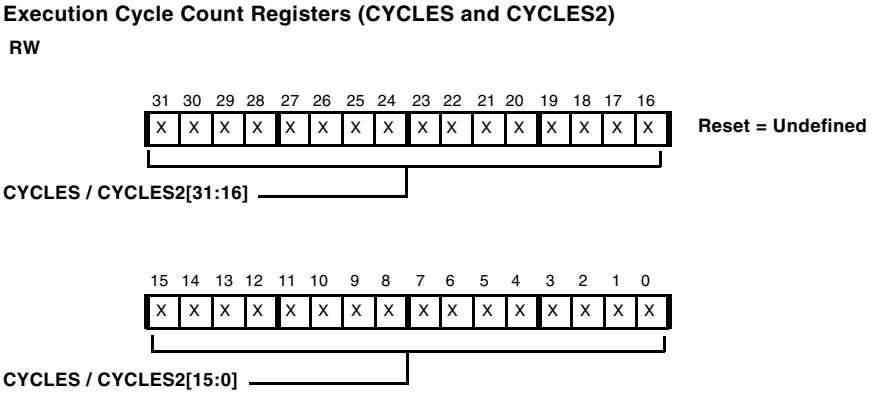


Figure 23-14. Execution Cycle Count Registers

Product Identification Register

The 32-bit DSP Device ID register (DSPID) is a core MMR that contains core identification and revision fields for the core.

DSPID Register

The DSP Device ID register (`DSPID`), shown in Figure 23-15, is a read-only register and is part of the core. The `Implementation` field contains the silicon revision of the device.

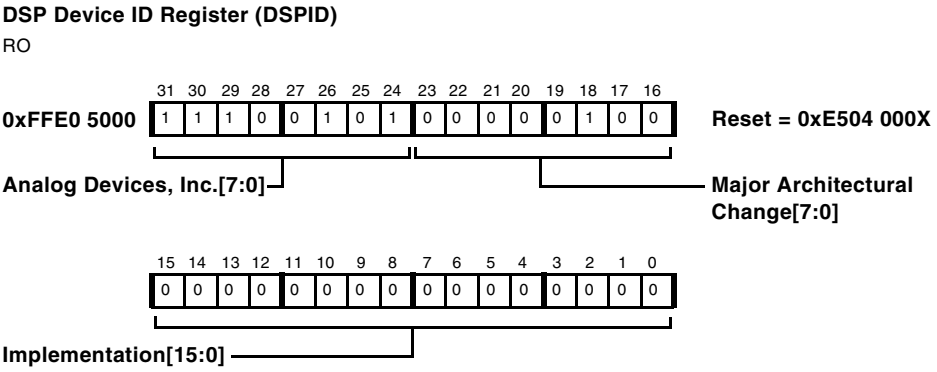


Figure 23-15. DSP Device ID Register

A BLACKFIN PROCESSOR CORE MMR ASSIGNMENTS

The Blackfin processor's memory-mapped registers (MMRs) are in the address range 0xFFE0 0000 – 0xFFFF FFFF.

All core MMRs must be accessed with a 32-bit read or write access.

This appendix lists core MMR addresses and register names. To find more information about an MMR, refer to the page shown in the "See Section" column. When viewing the PDF version of this document, click a reference in the "See Section" column to jump to additional information about the MMR.

L1 Data Memory Controller Registers

L1 data memory controller registers (0xFFE0 0000 – 0xFFE0 0404)

Table A-1. L1 Data Memory Controller Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 0004	DMEM_CONTROL	"Data Memory Control (DMEM_CONTROL) Register" on page 6-24
0xFFE0 0008	DCPLB_STATUS	"Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers" on page 6-59
0xFFE0 000C	DCPLB_FAULT_ADDR	"Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers" on page 6-60
0xFFE0 0100	DCPLB_ADDR0	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0104	DCPLB_ADDR1	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0108	DCPLB_ADDR2	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 010C	DCPLB_ADDR3	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0110	DCPLB_ADDR4	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0114	DCPLB_ADDR5	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0118	DCPLB_ADDR6	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 011C	DCPLB_ADDR7	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 0120	DCPLB_ADDR8	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0124	DCPLB_ADDR9	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0128	DCPLB_ADDR10	"WPDACNTn Registers" on page 23-11
0xFFE0 012C	DCPLB_ADDR11	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0130	DCPLB_ADDR12	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0134	DCPLB_ADDR13	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0138	DCPLB_ADDR14	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 013C	DCPLB_ADDR15	"Data CPLB Address (DCPLB_ADDRx) Registers" on page 6-56
0xFFE0 0200	DCPLB_DATA0	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0204	DCPLB_DATA1	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0 xFFE0 0208	DCPLB_DATA2	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 020C	DCPLB_DATA3	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0210	DCPLB_DATA4	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0214	DCPLB_DATA5	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0218	DCPLB_DATA6	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 021C	DCPLB_DATA7	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 0220	DCPLB_DATA8	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0224	DCPLB_DATA9	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0228	DCPLB_DATA10	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 022C	DCPLB_DATA11	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0230	DCPLB_DATA12	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0234	DCPLB_DATA13	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0238	DCPLB_DATA14	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 023C	DCPLB_DATA15	"Data CPLB Data (DCPLB_DATAx) Registers" on page 6-53
0xFFE0 0300	DTEST_COMMAND	"Data Test Command (DTEST_COMMAND) Register" on page 6-40
0xFFE0 0400	DTEST_DATA0	"Data Test Data (DTEST_DATA0) Register" on page 6-42
0xFFE0 0404	DTEST_DATA1	"Data Test Data (DTEST_DATA1) Register" on page 6-41

L1 Instruction Memory Controller Registers

L1 instruction memory controller registers (0xFFE0 1004 – 0xFFE0 1404)

Table A-2. L1 Instruction Memory Controller Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 1004	IMEM_CONTROL	"IMEM_CONTROL Register" on page 6-6
0xFFE0 1008	ICPLB_STATUS	"Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers" on page 6-59
0xFFE0 100C	ICPLB_FAULT_ADDR	"Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers" on page 6-60
0xFFE0 1100	ICPLB_ADDR0	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1104	ICPLB_ADDR1	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1108	ICPLB_ADDR2	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 110C	ICPLB_ADDR3	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1110	ICPLB_ADDR4	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1114	ICPLB_ADDR5	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1118	ICPLB_ADDR6	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 111C	ICPLB_ADDR7	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 1120	ICPLB_ADDR8	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1124	ICPLB_ADDR9	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1128	ICPLB_ADDR10	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 112C	ICPLB_ADDR11	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1130	ICPLB_ADDR12	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1134	ICPLB_ADDR13	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1138	ICPLB_ADDR14	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 113C	ICPLB_ADDR15	"Instruction CPLB Address (ICPLB_ADDRx) Registers" on page 6-58
0xFFE0 1200	ICPLB_DATA0	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1204	ICPLB_DATA1	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1208	ICPLB_DATA2	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 120C	ICPLB_DATA3	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1210	ICPLB_DATA4	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1214	ICPLB_DATA5	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1218	ICPLB_DATA6	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 121C	ICPLB_DATA7	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1220	ICPLB_DATA8	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1224	ICPLB_DATA9	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1228	ICPLB_DATA10	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 122C	ICPLB_DATA11	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1230	ICPLB_DATA12	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1234	ICPLB_DATA13	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1238	ICPLB_DATA14	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 123C	ICPLB_DATA15	"Instruction CPLB Data (ICPLB_DATAx) Registers" on page 6-51
0xFFE0 1300	ITEST_COMMAND	"Instruction Test Command (ITEST_COMMAND) Register" on page 6-21
0XFFE0 1400	ITEST_DATA0	"Instruction Test Data 0 (ITEST_DATA0) Register" on page 6-23
0XFFE0 1404	ITEST_DATA1	"Instruction Test Data (ITEST_DATA1) Register" on page 6-22

Interrupt Controller Registers

Interrupt controller registers (0xFFE0 2000 – 0xFFE0 2110)

Table A-3. Interrupt Controller Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 2000	EVT0 (EMU)	"Core Event Vector Table" on page 4-43
0xFFE0 2004	EVT1 (RST)	"Core Event Vector Table" on page 4-43
0xFFE0 2008	EVT2 (NMI)	"Core Event Vector Table" on page 4-43
0xFFE0 200C	EVT3 (EVX)	"Core Event Vector Table" on page 4-43
0xFFE0 2010	EVT4	"Core Event Vector Table" on page 4-43
0xFFE0 2014	EVT5 (IVHW)	"Core Event Vector Table" on page 4-43
0xFFE0 2018	EVT6 (TMR)	"Core Event Vector Table" on page 4-43
0xFFE0 201C	EVT7 (IVG7)	"Core Event Vector Table" on page 4-43
0xFFE0 2020	EVT8 (IVG8)	"Core Event Vector Table" on page 4-43
0xFFE0 2024	EVT9 (IVG9)	"Core Event Vector Table" on page 4-43
0xFFE0 2028	EVT10 (IVG10)	"Core Event Vector Table" on page 4-43
0xFFE0 202C	EVT11 (IVG11)	"Core Event Vector Table" on page 4-43
0xFFE0 2030	EVT12 (IVG12)	"Core Event Vector Table" on page 4-43

Table A-3. Interrupt Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 2034	EVT13 (IVG13)	"Core Event Vector Table" on page 4-43
0xFFE0 2038	EVT14 (IVG14)	"Core Event Vector Table" on page 4-43
0xFFE0 203C	EVT15 (IVG15)	"Core Event Vector Table" on page 4-43
0xFFE0 2104	IMASK	"System Interrupt Mask (SIC_IMASKx) Registers" on page 4-31
0xFFE0 2108	IPEND	"Core Interrupts Pending (IPEND) Register" on page 4-41
0xFFE0 210C	ILAT	"Core Interrupt Latch (ILAT) Register" on page 4-40
0xFFE0 2110	IPRIO	"Interrupt Priority Register and Write Buffer Depth" on page 6-36

Core Timer Registers

Core Timer registers (0xFFE0 3000 – 0xFFE0 300C)

Table A-4. Core Timer Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 3000	TCNTL	"TCNTL Register" on page 16-48
0xFFE0 3004	TPERIOD	"TPERIOD Register" on page 16-50
0xFFE0 3008	TSCALE	"TSCALE Register" on page 16-51
0xFFE0 300C	TCOUNT	"TCOUNT Register" on page 16-50

DSP Identification Register

Table A-5. DSP Identification Register

Memory-mapped Address	Register Name	See Section
0xFFE0 5000	DSPID	"DSPID Register" on page 23-27

Trace Unit Registers

Trace Unit registers (0xFFE0 6000 – 0xFFE0 6100)

Table A-6. Trace Unit Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 6000	TBUFCTL	"TBUFCTL Register" on page 23-16
0xFFE0 6004	TBUFSTAT	"TBUFSTAT Register" on page 23-17
0xFFE0 6100	TBUF	"TBUF Register" on page 23-18

Watchpoint and Patch Registers

Watchpoint and Patch registers (0xFFE0 7000 – 0xFFE0 7200)

Table A-7. Watchpoint and Patch Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 7000	WPIACTL	"WPIACTL Register" on page 23-8
0xFFE0 7040	WPIA0	"WPIAn Registers" on page 23-6
0xFFE0 7044	WPIA1	"WPIAn Registers" on page 23-6
0xFFE0 7048	WPIA2	"WPIAn Registers" on page 23-6
0xFFE0 704C	WPIA3	"WPIAn Registers" on page 23-6
0xFFE0 7050	WPIA4	"WPIAn Registers" on page 23-6
0xFFE0 7054	WPIA5	"WPIAn Registers" on page 23-6
0xFFE0 7080	WPIACNT0	"WPIACNTn Registers" on page 23-7
0xFFE0 7084	WPIACNT1	"WPIACNTn Registers" on page 23-7
0xFFE0 7088	WPIACNT2	"WPIACNTn Registers" on page 23-7
0xFFE0 708C	WPIACNT3	"WPIACNTn Registers" on page 23-7
0xFFE0 7090	WPIACNT4	"WPIACNTn Registers" on page 23-7
0xFFE0 7094	WPIACNT5	"WPIACNTn Registers" on page 23-7
0xFFE0 7100	WPDACTL	"WPDACTL Register" on page 23-12
0xFFE0 7140	WPDA0	"WPDAn Registers" on page 23-10
0xFFE0 7144	WPDA1	"WPDAn Registers" on page 23-10
0xFFE0 7180	WPDACNT0	"WPDACNTn Registers" on page 23-11
0xFFE0 7184	WPDACNT1	"WPDACNTn Registers" on page 23-11
0xFFE0 7200	WPSTAT	"WPSTAT Register" on page 23-12

Performance Monitor Registers

Performance Monitor registers (0xFFE0 8000 – 0xFFE0 8104)

Table A-8. Performance Monitor Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 8000	PFCTL	"PFCTL Register" on page 23-21
0xFFE0 8100	PFCNTR0	"PFCNTRn Registers" on page 23-20
0xFFE0 8104	PFCNTR1	"PFCNTRn Registers" on page 23-20

B SYSTEM MMR ASSIGNMENTS

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF FFFF.
- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.
- All system MMR space that is not defined in this appendix is reserved for internal use only.

This appendix lists MMR addresses and register names. To find more information about an MMR, refer to the page shown in the “See Section” column. When viewing the PDF version of this document, click a reference in the “See Section” column to jump to additional information about the MMR.

Dynamic Power Management Registers

Dynamic Power Management registers (0xFFC0 0000 – 0xFFC0 00FF)

Table B-1. Dynamic Power Management Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0000	PLL_CTL	"PLL Control (PLL_CTL) Register" on page 8-7
0xFFC0 0004	PLL_DIV	"PLL Divide (PLL_DIV) Register" on page 8-7
0xFFC0 0008	VR_CTL	"Voltage Regulator Control (VR_CTL) Register" on page 8-26
0xFFC0 000C	PLL_STAT	"PLL Status (PLL_STAT) Register" on page 8-10
0xFFC0 0010	PLL_LOCKCNT	"PLL Lock Count (PLL_LOCKCNT) Register" on page 8-11

System Reset and Interrupt Control Registers

System Reset and Interrupt Controller registers (0xFFC0 0100 – 0xFFC0 01FF)

Table B-2. System Interrupt Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0100	SWRST	"SWRST Register" on page 3-16
0xFFC0 0104	SYSCR	"SYSCR Register" on page 3-14
0xFFC0 0108	SIC_RVECT	NA
0xFFC0 010C 0xFFC0 0128	SIC_IMASK 0 SIC_IMASK 1	"System Interrupt Mask (SIC_IMASKx) Registers" on page 4-31

Table B-2. System Interrupt Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0110 0xFFC0 0114 0xFFC0 0118 0xFFC0 011C 0xFFC0 0134 0xFFC0 0138 0xFFC0 013C	SIC_IAR0 SIC_IAR1 SIC_IAR2 SIC_IAR3 SIC_IAR4 SIC_IAR5 SIC_IAR6	"System Interrupt Assignment (SIC_IARx) Registers" on page 4-33
0xFFC0 0120 0xFFC0 012C	SIC_ISR0 SIC_ISR1	"System Interrupt Status (SIC_ISRx) Registers" on page 4-28
0xFFC0 0124 0xFFC0 0130	SIC_IWR0 SIC_IWR1	"System Interrupt Wakeup-Enable (SIC_IWRx) Registers" on page 4-26

Watchdog Timer Registers

Watchdog Timer registers (0xFFC0 0200 – 0xFFC0 02FF)

Table B-3. Watchdog Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0200	WDOG_CTL	"WDOG_CTL Register" on page 16-55
0xFFC0 0204	WDOG_CNT	"WDOG_CNT Register" on page 16-52
0xFFC0 0208	WDOG_STAT	"WDOG_STAT Register" on page 16-53

Real-Time Clock Registers

Real-Time Clock registers (0xFFC0 0300 – 0xFFC0 03FF)

Table B-4. Real-Time Clock Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0300	RTC_STAT	"RTC_STAT Register" on page 17-13
0xFFC0 0304	RTC_ICTL	"RTC_ICTL Register" on page 17-15
0xFFC0 0308	RTC_ISTAT	"RTC_ISTAT Register" on page 17-16
0xFFC0 030C	RTC_SWCNT	"RTC_SWCNT Register" on page 17-17
0xFFC0 0310	RTC_ALARM	"RTC_ALARM Register" on page 17-18
0xFFC0 0314	RTC_PREN	"RTC_PREN Register" on page 17-19

UART Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF)
UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF)
UART2 controller registers (0xFFC0 2100 – 0xFFC0 21FF)

Table B-5. UART0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0400	UART0_THR	"UART Transmit Holding (UARTx_THR) Register" on page 13-6
0xFFC0 0400	UART0_RBR	"UART Receive Buffer (UARTx_RBR) Register" on page 13-7
0xFFC0 0400	UART0_DLL	"UARTx_DLL and UARTx_DLH Registers" on page 13-12
0xFFC0 0404	UART0_DLH	"UARTx_DLL and UARTx_DLH Registers" on page 13-12

Table B-5. UART0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0404	UART0_IER	"UART Interrupt Enable (UARTx_IER) Register" on page 13-8
0xFFC0 0408	UART0_IIR	"UART Interrupt Identification (UARTx_IIR) Register" on page 13-10
0xFFC0 040C	UART0_LCR	"UART Line Control (UARTx_LCR) Register" on page 13-4
0xFFC0 0410	UART0_MCR	"UART Modem Control (UARTx_MCR) Register" on page 13-4
0xFFC0 0414	UART0_LSR	"UART Line Status (UARTx_LSR) Register" on page 13-5
0xFFC0 041C	UART0_SCR	"UART Scratch (UARTx_SCR) Register" on page 13-14
0xFFC0 0424	UART0_GCTL	"UART Global Control (UARTx_GCTL) Register" on page 13-14

Table B-6. UART1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC 02000	UART1_THR	"UART Transmit Holding (UARTx_THR) Register" on page 13-6
0xFFC 02000	UART1_RBR	"UART Receive Buffer (UARTx_RBR) Register" on page 13-7
0xFFC 02000	UART1_DLL	"UARTx_DLL and UARTx_DLH Registers" on page 13-12
0xFFC0 2004	UART1_DLH	"UARTx_DLL and UARTx_DLH Registers" on page 13-12
0xFFC0 2004	UART1_IER	"UART Interrupt Enable (UARTx_IER) Register" on page 13-8
0xFFC0 2008	UART1_IIR	"UART Interrupt Identification (UARTx_IIR) Register" on page 13-10
0xFFC0 200C	UART1_LCR	"UART Line Control (UARTx_LCR) Register" on page 13-4
0xFFC0 2010	UART1_MCR	"UART Modem Control (UARTx_MCR) Register" on page 13-4
0xFFC0 2014	UART1_LSR	"UART Line Status (UARTx_LSR) Register" on page 13-5

Table B-6. UART1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 201C	UART1_SCR	"UART Scratch (UARTx_SCR) Register" on page 13-14
0xFFC0 2024	UART1_GCTL	"UART Global Control (UARTx_GCTL) Register" on page 13-14

Table B-7. UART2 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2100	UART2_THR	"UART Transmit Holding (UARTx_THR) Register" on page 13-6
0xFFC0 2100	UART2_RBR	"UART Receive Buffer (UARTx_RBR) Register" on page 13-7
0xFFC0 2100	UART2_DLL	"UARTx_DLL and UARTx_DLH Registers" on page 13-12
0xFFC0 2104	UART2_DLH	"UARTx_DLL and UARTx_DLH Registers" on page 13-12
0xFFC0 2104	UART2_IER	"UART Interrupt Enable (UARTx_IER) Register" on page 13-8
0xFFC0 2108	UART2_IIR	"UART Interrupt Identification (UARTx_IIR) Register" on page 13-10
0xFFC0 210C	UART2_LCR	"UART Line Control (UARTx_LCR) Register" on page 13-4
0xFFC0 2110	UART2_MCR	"UART Modem Control (UARTx_MCR) Register" on page 13-4
0xFFC0 2114	UART2_LSR	"UART Line Status (UARTx_LSR) Register" on page 13-5
0xFFC0 211C	UART2_SCR	"UART Scratch (UARTx_SCR) Register" on page 13-14
0xFFC0 2124	UART2_GCTL	"UART Global Control (UARTx_GCTL) Register" on page 13-14

SPI Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF)
SPI1 controller registers (0xFFC0 2300 – 0xFFC0 23FF)
SPI2 controller registers (0xFFC0 2400 – 0xFFC0 024FF)

Table B-8. SPI0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0500	SPI0_CTL	"SPI Control (SPIx_CTL) Register" on page 10-9
0xFFC0 0504	SPI0_FLG	"SPI Flag (SPIx_FLG) Register" on page 10-10
0xFFC0 0508	SPI0_STAT	"SPI Status (SPIx_STAT) Register" on page 10-17
0xFFC0 050C	SPI0_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 10-18
0xFFC0 0510	SPI0_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 10-19
0xFFC0 0514	SPI0_BAUD	"SPI BAUD Rate (SPIx_BAUD) Register" on page 10-8
0xFFC0 0518	SPI0_SHADOW	"SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register" on page 10-20

Table B-9. SPI1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2300	SPI1_CTL	"SPI Control (SPIx_CTL) Register" on page 10-9
0xFFC0 2304	SPI1_FLG	"SPI Flag (SPIx_FLG) Register" on page 10-10
0xFFC0 2308	SPI1_STAT	"SPI Status (SPIx_STAT) Register" on page 10-17
0xFFC0 230C	SPI1_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 10-18
0xFFC0 2310	SPI1_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 10-19

Table B-9. SPI1 Controller Registers (Cont'd) (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2314	SPI1_BAUD	"SPI BAUD Rate (SPIx_BAUD) Register" on page 10-8
0xFFC0 2318	SPI1_SHADOW	"SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register" on page 10-20

Table B-10. SPI2 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2400	SPI2_CTL	"SPI Control (SPIx_CTL) Register" on page 10-9
0xFFC0 2404	SPI2_FLG	"SPI Flag (SPIx_FLG) Register" on page 10-10
0xFFC0 2408	SPI2_STAT	"SPI Status (SPIx_STAT) Register" on page 10-17
0xFFC0 240C	SPI2_TDBR	"SPI Transmit Data Buffer (SPIx_TDBR) Register" on page 10-18
0xFFC0 2410	SPI2_RDBR	"SPI Receive Data Buffer (SPIx_RDBR) Register" on page 10-19
0xFFC0 2414	SPI2_BAUD	"SPI BAUD Rate (SPIx_BAUD) Register" on page 10-8
0xFFC0 2418	SPI2_SHADOW	"SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register" on page 10-20

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF)

Table B-11. Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0600	TIMER0_CONFIG	"TIMERx_CONFIG Registers" on page 16-8
0xFFC0 0604	TIMER0_COUNTER	"TIMERx_COUNTER Registers" on page 16-9

Table B-11. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0608	TIMER0_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 060C	TIMER0_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 0610	TIMER1_CONFIG	"TIMERx_CONFIG Registers" on page 16-8
0xFFC0 0614	TIMER1_COUNTER	"TIMERx_COUNTER Registers" on page 16-9
0xFFC0 0618	TIMER1_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 061C	TIMER1_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 0620	TIMER2_CONFIG	"TIMERx_CONFIG Registers" on page 16-8
0xFFC0 0624	TIMER2_COUNTER	"TIMERx_COUNTER Registers" on page 16-9
0xFFC0 0628	TIMER2_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 062C	TIMER2_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 16-10
0xFFC0 0640	TIMER_ENABLE	"TIMER_ENABLE Register" on page 16-4
0xFFC0 0644	TIMER_DISABLE	"TIMER_DISABLE Register" on page 16-5
0xFFC0 0648	TIMER_STATUS	"TIMER_STATUS Register" on page 16-6

Programmable Flag Registers

Programmable Flag registers (0xFFC0 0700 – 0xFFC0 07FF)

Table B-12. Programmable Flag Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0700	FIO_FLAG_D	"Flag Data (FIO_FLAG_D) Register" on page 14-8
0xFFC0 0704	FIO_FLAG_C	"Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) Registers" on page 14-8
0xFFC0 0708	FIO_FLAG_S	"Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) Registers" on page 14-8
0xFFC0 070C	FIO_FLAG_T	"Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) Registers" on page 14-8
0xFFC0 0710	FIO_MASKA_D	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0714	FIO_MASKA_C	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0718	FIO_MASKA_S	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 071C	FIO_MASKA_T	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0720	FIO_MASKB_D	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0724	FIO_MASKB_C	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0728	FIO_MASKB_S	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14

Table B-12. Programmable Flag Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 072C	FIO_MASKB_T	"Flag Interrupt A (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T) Registers" on page 14-14
0xFFC0 0730	FIO_DIR	"Flag Direction (FIO_DIR) Register" on page 14-5
0xFFC0 0734	FIO_POLAR	"Flag Polarity (FIO_POLAR) Register" on page 14-18
0xFFC0 0738	FIO_EDGE	"Flag Interrupt Sensitivity (FIO_EDGE) Register" on page 14-18
0xFFC0 073C	FIO_BOTH	"Flag Set on Both Edges (FIO_BOTH) Register" on page 14-19
0xFFC0 0740	FIO_INEN	"Flag Input Enable (FIO_INEN) Register" on page 14-20

SPORT Controller Registers

SPORT0 registers (0xFFC0 0800 – 0xFFC0 08FF)
SPORT1 registers (0xFFC0 0900 – 0xFFC0 09FF)
SPORT2 registers (0xFFC0 2500 – 0xFFC0 25FF)
SPORT3 registers (0xFFC0 2600 – 0xFFC0 26FF)

Table B-13. SPORT0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0800	SPORT0_TCR1	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 0804	SPORT0_TCR2	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 0808	SPORT0_TCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29

Table B-13. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 080C	SPORT0_TFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 0810	SPORT0_TX	"SPORT Transmit Data (SPORTx_TX) Register" on page 12-22
0xFFC0 0818	SPORT0_RX	"SPORT Receive Data (SPORTx_RX) Register" on page 12-24
0xFFC0 0820	SPORT0_RCR1	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 0824	SPORT0_RCR2	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 0828	SPORT0_RCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 082C	SPORT0_RFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 0830	SPORT0_STAT	"SPORT Status (SPORTx_STAT) Register" on page 12-26
0xFFC0 0834	SPORT0_CHNL	"SPORT Current Channel (SPORTx_CHNL) Register" on page 12-59
0xFFC0 0838	SPORT0_MCMC1	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 083C	SPORT0_MCMC2	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 0840	SPORT0_MTCS0	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0844	SPORT0_MTCS1	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0848	SPORT0_MTCS2	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64

Table B-13. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 084C	SPORT0_MTCS3	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0850	SPORT0_MRCS0	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 0854	SPORT0_MRCS1	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 0858	SPORT0_MRCS2	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 085C	SPORT0_MRCS3	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62

Table B-14. SPORT1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0900	SPORT1_TCR1	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 0904	SPORT1_TCR2	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 0908	SPORT1_TCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 090C	SPORT1_TFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 0910	SPORT1_TX	"SPORT Transmit Data (SPORTx_TX) Register" on page 12-22
0xFFC0 0918	SPORT1_RX	"SPORT Receive Data (SPORTx_RX) Register" on page 12-24
0xFFC0 0920	SPORT1_RCR1	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16

Table B-14. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0924	SPORT1_RCR2	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 0928	SPORT1_RCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 092C	SPORT1_RFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 0930	SPORT1_STAT	"SPORT Status (SPORTx_STAT) Register" on page 12-26
0xFFC0 0934	SPORT1_CHNL	"SPORT Current Channel (SPORTx_CHNL) Register" on page 12-59
0xFFC0 0938	SPORT1_MCMC1	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 093C	SPORT1_MCMC2	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 0940	SPORT1_MTCS0	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0944	SPORT1_MTCS1	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0948	SPORT1_MTCS2	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 094C	SPORT1_MTCS3	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 0950	SPORT1_MRCS0	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 0954	SPORT1_MRCS1	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62

Table B-14. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0958	SPORT1_MRCS2	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 095C	SPORT1_MRCS3	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62

Table B-15. SPORT2 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2500	SPORT2_TCR1	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 2504	SPORT2_TCR2	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 2508	SPORT2_TCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 250C	SPORT2_TFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 2510	SPORT2_TX	"SPORT Transmit Data (SPORTx_TX) Register" on page 12-22
0xFFC0 2518	SPORT2_RX	"SPORT Receive Data (SPORTx_RX) Register" on page 12-24
0xFFC0 2520	SPORT2_RCR1	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 2524	SPORT2_RCR2	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 2528	SPORT2_RCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29

Table B-15. SPORT2 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 252C	SPORT2_RFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 2530	SPORT2_STAT	"SPORT Status (SPORTx_STAT) Register" on page 12-26
0xFFC0 2534	SPORT2_CHNL	"SPORT Current Channel (SPORTx_CHNL) Register" on page 12-59
0xFFC0 2538	SPORT2_MCMC1	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 253C	SPORT2_MCMC2	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 2540	SPORT2_MTCS0	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2544	SPORT2_MTCS1	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2548	SPORT2_MTCS2	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 254C	SPORT2_MTCS3	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2550	SPORT2_MRCS0	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 2554	SPORT2_MRCS1	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 2558	SPORT2_MRCS2	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 255C	SPORT2_MRCS3	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62

Table B-16. SPORT3 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2600	SPORT3_TCR1	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 2604	SPORT3_TCR2	"SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers" on page 12-11
0xFFC0 2608	SPORT3_TCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 260C	SPORT3_TFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 2610	SPORT3_TX	"SPORT Transmit Data (SPORTx_TX) Register" on page 12-22
0xFFC0 2618	SPORT3_RX	"SPORT Receive Data (SPORTx_RX) Register" on page 12-24
0xFFC0 2620	SPORT3_RCR1	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 2624	SPORT3_RCR2	"SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers" on page 12-16
0xFFC0 2628	SPORT3_RCLKDIV	"SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers" on page 12-29
0xFFC0 262C	SPORT3_RFSDIV	"SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register" on page 12-30
0xFFC0 2630	SPORT3_STAT	"SPORT Status (SPORTx_STAT) Register" on page 12-26
0xFFC0 2634	SPORT3_CHNL	"SPORT Current Channel (SPORTx_CHNL) Register" on page 12-59
0xFFC0 2638	SPORT3_MCMC1	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51
0xFFC0 263C	SPORT3_MCMC2	"SPORT Multichannel Configuration (SPORTx_MCMCn) Registers" on page 12-51

Table B-16. SPORT3 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2640	SPORT3_MTCS0	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2644	SPORT3_MTCS1	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2648	SPORT3_MTCS2	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 264C	SPORT3_MTCS3	"SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers" on page 12-64
0xFFC0 2650	SPORT3_MRCS0	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 2654	SPORT3_MRCS1	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 2658	SPORT3_MRCS2	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62
0xFFC0 265C	SPORT3_MRCS3	"SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers" on page 12-62

External Bus Interface Unit Registers

External Bus Interface Unit registers (0xFFC0 0A00 – 0xFFC0 0AFF)

Table B-17. External Bus Interface Unit Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0A00	EBIU_AMGCTL	"EBIU_AMGCTL Register" on page 18-11
0xFFC0 0A04	EBIU_AMBCTL0	"EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers" on page 18-13
0xFFC0 0A08	EBIU_AMBCTL1	"EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers" on page 18-13

Table B-17. External Bus Interface Unit Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0A10	EBIU_SDGCTL	"EBIU_SDGCTL Register" on page 18-36
0xFFC0 0A14	EBIU_SDBCTL	"EBIU_SDBCTL Register" on page 18-48
0xFFC0 0A18	EBIU_SDRRC	"EBIU_SDRRC Register" on page 18-52
0xFFC0 0A1C	EBIU_SDSTAT	"EBIU_SDSTAT Register" on page 18-51

DMA/Memory DMA Control Registers

DMA Controller 0 registers (0xFFC0 0C00 – 0xFFC0 0FFF)
DMA Controller 1 registers (0xFFC0 1C00 – 0xFFC0 1FFF)

Table B-18. DMA Traffic Control Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0B0C 0xFFC0 1B0C	DMAC0_TC_PER DMAC1_TC_PER	"DMA Traffic Control Counter Period (DMACx_TC_PER) Register and DMA Traffic Control Counter (DMACx_TC_CNT) Register" on page 9-65
0xFFC0 0B10 0xFFC0 1B10	DMAC0_TC_CNT DMAC1_TC_CNT	"DMA Traffic Control Counter Period (DMACx_TC_PER) Register and DMA Traffic Control Counter (DMACx_TC_CNT) Register" on page 9-65

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table B-19 and Table B-19. Table B-19 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table B-21 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA channel 0 Y modify register is called DMA0_Y_MODIFY, and its address is 0xFFC0 0C1C. Likewise, the memory DMA stream 0 source current address register is called MDMA_S0_CURR_ADDR, and its address is 0xFFC0 0E64.

Table B-19. DMA Channel Base Addresses for DMA Controller 0

DMA Channel Identifier	MMR Base Address	Register Prefix
DMA0	0xFFC0 0C00	DMA0_
DMA1	0xFFC0 0C40	DMA1_
DMA2	0xFFC0 0C80	DMA2_
DMA3	0xFFC0 0CC0	DMA3_
DMA4	0xFFC0 0D00	DMA4_
DMA5	0xFFC0 0D40	DMA5_
DMA6	0xFFC0 0D80	DMA6_
DMA7	0xFFC0 0DC0	DMA7_
Mem DMA 0 Stream 0 Destination	0xFFC0 0E00	MDMA0_D0_
Mem DMA 0 Stream 0 Source	0xFFC0 0E40	MDMA0_S0_
Mem DMA 0 Stream 1 Destination	0xFFC0 0E80	MDMA0_D1_
Mem DMA 0 Stream 1 Source	0xFFC0 0EC0	MDMA0_S1_

Table B-20. DMA Channel Base Addresses for DMA Controller 1

DMA Channel Identifier	MMR Base Address	Register Prefix
DMA8	0xFFC0 1C00	DMA8_
DMA9	0xFFC0 1C40	DMA9_
DMA10	0xFFC0 1C80	DMA10_
DMA11	0xFFC0 1CC0	DMA11_
DMA12	0xFFC0 1D00	DMA12_

Table B-20. DMA Channel Base Addresses for DMA Controller 1 (Cont'd)

DMA Channel Identifier	MMR Base Address	Register Prefix
DMA13	0xFFC0 1D40	DMA13_
DMA14	0xFFC0 1D80	DMA14_
DMA15	0xFFC0 1DC0	DMA15_
DMA16	0xFFC0 1E00	DMA16_
DMA17	0xFFC0 1E40	DMA17_
DMA18	0xFFC0 1E80	DMA18_
DMA19	0xFFC0 1EC0	DMA19_
Mem DMA 1 Stream 0 Destination	0xFFC0 1F00	MDMA1_D0_
Mem DMA 1 Stream 0 Source	0xFFC0 1F40	MDMA1_S0_
Mem DMA 1 Stream 1 Destination	0xFFC0 1F80	MDMA1_D1_
Mem DMA 1 Stream 1 Source	0xFFC0 1FC0	MDMA1_S1_

Table B-21. DMA Register Suffix and Offset

Register Suffix	Offset From Base	See Section
NEXT_DESC_PTR	0x00	"Next Descriptor Pointer (DMAx_NEXT_DESC_PTR / MDMAx_yy_NEXT_DESC_PTR) Registers" on page 9-10
START_ADDR	0x04	"Start Address Register (DMAx_START_ADDR/MDMAx_yy_START_ADDR)" on page 9-11
CONFIG	0x08	"DMA Configuration Register (DMAx_CONFIG / MDMAx_yy_CONFIG)" on page 9-12
X_COUNT	0x10	"Inner Loop Count (DMAx_X_COUNT, MDMAx_yy_X_COUNT) Registers" on page 9-15
X_MODIFY	0x14	"Inner Loop Address Increment (DMAx_X_MODIFY, MDMAx_yy_X_MODIFY) Registers" on page 9-16

Table B-21. DMA Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	See Section
Y_COUNT	0x18	"Outer Loop Count (DMAx_Y_COUNT, MDMAx_yy_Y_COUNT) Registers" on page 9-17
Y_MODIFY	0x1C	"Outer Loop Address Increment (DMAx_Y_MODIFY, MDMAx_yy_Y_MODIFY) Registers" on page 9-17
CURR_DESC_PTR	0x20	"Current Descriptor Pointer (DMAx_CURR_DESC_PTR, MDMAx_yy_CURR_DESC_PTR) Registers" on page 9-18
CURR_ADDR	0x24	"Current Address (DMAx_CURR_ADDR, MDMAx_yy_CURR_ADDR) Registers" on page 9-19
IRQ_STATUS	0x28	"Interrupt Status Register (DMAx_IRQ_STATUS / MDMAx_yy_IRQ_STATUS)" on page 9-28
PERIPHERAL_MAP	0x2C	"Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers" on page 9-22
CURR_X_COUNT	0x30	"Current Inner Loop Count (DMAx_CURR_X_COUNT, MDMAx_yy_CURR_X_COUNT) Registers" on page 9-20
CURR_Y_COUNT	0x38	"Current Outer Loop Count Register (DMAx_CURR_Y_COUNT / MDMAx_yy_CURR_Y_COUNT)" on page 9-21

Parallel Peripheral Interface (PPI) Registers

Parallel Peripheral Interface (PPI) registers (0xFFC0 1000 – 0xFFC0 10FF)

Table B-22. Parallel Peripheral Interface (PPI) Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1000	PPI_CONTROL	"PPI_CONTROL Register" on page 11-3
0xFFC0 1004	PPI_STATUS	"PPI_STATUS Register" on page 11-8
0xFFC0 1008	PPI_COUNT	"PPI_COUNT Register" on page 11-10
0xFFC0 100C	PPI_DELAY	"PPI_DELAY Register" on page 11-10
0xFFC0 1010	PPI_FRAME	"PPI_FRAME Register" on page 11-11

Two-Wire Interface Registers

TWI0 registers (0xFFC0 1400 – 0xFFC0 14FF)
TWI1 registers (0xFFC0 2200 – 0xFFC0 22FF)

Table B-23. TWI0 Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1400	TWI0_CLKDIV	"TWI Clock Divider (TWIx_CLKDIV) Registers" on page 20-6
0xFFC0 1404	TWI0_CONTROL	"TWI Control (TWIx_CONTROL) Registers" on page 20-5
0xFFC0 1408	TWI0_SLAVE_CTRL	"TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers" on page 20-7

Table B-23. TWI0 Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 140C	TWI0_SLAVE_STAT	"TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers" on page 20-10
0xFFC0 1410	TWI0_SLAVE_ADDR	"TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers" on page 20-9
0xFFC0 1414	TWI0_MASTER_CTRL	"TWI Master Mode Control (TWIx_MASTER_CTRL) Registers" on page 20-11
0xFFC0 1418	TWI0_MASTER_STAT	"TWI Master Mode Status (TWIx_MASTER_STAT) Registers" on page 20-15
0xFFC0 141C	TWI0_MASTER_ADDR	"TWI Master Mode Address (TWIx_MASTER_ADDR) Registers" on page 20-14
0xFFC0 1420	TWI0_INT_STAT	"TWI Interrupt Status (TWIx_INT_STAT) Registers" on page 20-25
0xFFC0 1428	TWI0_FIFO_CTRL	"TWI FIFO Control (TWIx_FIFO_CTRL) Registers" on page 20-19
0xFFC0 142C	TWI0_FIFO_STAT	"TWI FIFO Status (TWIx_FIFO_STAT) Registers" on page 20-21
0xFFC0 1480	TWI0_XMT_DATA8	"TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers" on page 20-28
0xFFC0 1484	TWI0_XMT_DATA16	"TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers" on page 20-29
0xFFC0 1488	TWI0_RCV_DATA8	"TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers" on page 20-30
0xFFC0 148C	TWI0_RCV_DATA16	"TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers" on page 20-31

Table B-24. TWI1 Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2200	TWI1_CLKDIV	"TWI Clock Divider (TWIx_CLKDIV) Registers" on page 20-6
0xFFC0 2204	TWI1_CONTROL	"TWI Control (TWIx_CONTROL) Registers" on page 20-5
0xFFC0 2208	TWI1_SLAVE_CTRL	"TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers" on page 20-7
0xFFC0 220C	TWI1_SLAVE_STAT	"TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers" on page 20-10
0xFFC0 2210	TWI1_SLAVE_ADDR	"TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers" on page 20-9
0xFFC0 2214	TWI1_MASTER_CTRL	"TWI Master Mode Control (TWIx_MASTER_CTRL) Registers" on page 20-11
0xFFC0 2218	TWI1_MASTER_STAT	"TWI Master Mode Status (TWIx_MASTER_STAT) Registers" on page 20-15
0xFFC0 221C	TWI1_MASTER_ADDR	"TWI Master Mode Address (TWIx_MASTER_ADDR) Registers" on page 20-14
0xFFC0 2220	TWI1_INT_STAT	"TWI Interrupt Status (TWIx_INT_STAT) Registers" on page 20-25
0xFFC0 2228	TWI1_FIFO_CTRL	"TWI FIFO Control (TWIx_FIFO_CTRL) Registers" on page 20-19
0xFFC0 222C	TWI1_FIFO_STAT	"TWI FIFO Status (TWIx_FIFO_STAT) Registers" on page 20-21
0xFFC0 2280	TWI1_XMT_DATA8	"TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers" on page 20-28
0xFFC0 2284	TWI1_XMT_DATA16	"TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers" on page 20-29

Table B-24. TWI1 Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2288	TWI1_RCV_DATA8	"TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers" on page 20-30
0xFFC0 228C	TWI1_RCV_DATA16	"TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers" on page 20-31

GPIO Ports C, D, and E Registers

GPIO Ports C, D, E registers (0xFFC0 1500 – 0xFFC0 15FF)

Table B-25. GPIO Port C Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1500	GPIO_C_CNFG	"GPIO Configuration (GPIO_x_CNFG) Register" on page 15-5
0xFFC0 1510	GPIO_C_D	"GPIO Data (GPIO_x_D) Register" on page 15-12
0xFFC0 1520 0xFFC0 1530 0xFFC0 1540	GPIO_C_C, GPIO_C_S, GPIO_C_T	"GPIO Clear (GPIO_x_C), GPIO Set (GPIO_x_S), and GPIO Toggle (GPIO_x_T) Registers" on page 15-14
0xFFC0 1550	GPIO_C_DIR	"GPIO Direction (GPIO_x_DIR) Register" on page 15-7
0xFFC0 1560	GPIO_C_INEN	"GPIO Input Enable (GPIO_x_INEN) Register" on page 15-9

Table B-26. GPIO Port D Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1504	GPIO_D_CNFG	"GPIO Configuration (GPIO_x_CNFG) Register" on page 15-5
0xFFC0 1514	GPIO_D_D	"GPIO Data (GPIO_x_D) Register" on page 15-12

Table B-26. GPIO Port D Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1524 0xFFC0 1534 0xFFC0 1544	GPIO_D_C, GPIO_D_S, GPIO_D_T	"GPIO Clear (GPIO_x_C), GPIO Set (GPIO_x_S), and GPIO Toggle (GPIO_x_T) Registers" on page 15-14
0xFFC0 1554	GPIO_D_DIR	"GPIO Direction (GPIO_x_DIR) Register" on page 15-7
0xFFC0 1564	GPIO_D_INEN	"GPIO Input Enable (GPIO_x_INEN) Register" on page 15-9

Table B-27. GPIO Port E Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1508	GPIO_E_CNFG	"GPIO Configuration (GPIO_x_CNFG) Register" on page 15-5
0xFFC0 1518	GPIO_E_D	"GPIO Data (GPIO_x_D) Register" on page 15-12
0xFFC0 1528 0xFFC0 1538 0xFFC0 1548	GPIO_E_C, GPIO_E_S, GPIO_E_T	"GPIO Clear (GPIO_x_C), GPIO Set (GPIO_x_S), and GPIO Toggle (GPIO_x_T) Registers" on page 15-14
0xFFC0 1558	GPIO_E_DIR	"GPIO Direction (GPIO_x_DIR) Register" on page 15-7
0xFFC0 1568	GPIO_E_INEN	"GPIO Input Enable (GPIO_x_INEN) Register" on page 15-9

MXVR Interface Registers

MXVR Interface registers (0xFFC0 2700–28FF).

Table B-28. MXVR Interface Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2700	MXVR_CONFIG	"MXVR Configuration Register (MXVR_CONFIG)" on page 21-5
0xFFC0 2704	MXVR_PLL_CTL_0	"MXVR PLL Control Register 0 (MXVR_PLL_CTL_0)" on page 21-11
0xFFC0 2708 0xFFC0 270C	MXVR_STATE_0 MXVR_STATE_1	"MXVR State Registers (MXVR_STATE_0, MXVR_STATE_1)" on page 21-20
0xFFC0 2710	MXVR_INT_STAT_0	"MXVR Interrupt Status Register 0 (MXVR_INT_STAT_0)" on page 21-31
0xFFC0 2714	MXVR_INT_STAT_1	"MXVR Interrupt Status Register_1 (MXVR_INT_STAT_1)" on page 21-42
0xFFC0 2718	MXVR_INT_EN_0	"MXVR Interrupt Enable Register 0 (MXVR_INT_EN_0)" on page 21-46
0xFFC0 271C	MXVR_INT_EN_1	"MXVR Interrupt Enable Register 1 (MXVR_INT_EN_1)" on page 21-48
0xFFC0 2720	MXVR_POSITION	"MXVR Node Position Register (MXVR_POSITION)" on page 21-50
0xFFC0 2724	MXVR_MAX_POSITION	"MXVR Maximum Node Position Register (MXVR_MAX_POSITION)" on page 21-51
0xFFC0 2728	MXVR_DELAY	"MXVR Node Frame Delay Register (MXVR_DELAY)" on page 21-53
0xFFC0 272C	MXVR_MAX_DELAY	"MXVR Maximum Node Frame Delay Register (MXVR_MAX_DELAY)" on page 21-55
0xFFC0 2730	MXVR_LADDR	"MXVR Logical Address Register (MXVR_LADDR)" on page 21-57

Table B-28. MXVR Interface Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2734	MXVR_GADDR	"MXVR Group Address Register (MXVR_GADDR)" on page 21-59
0xFFC0 2738	MXVR_AADDR	"MXVR Alternate Address Register (MXVR_AADDR)" on page 21-59
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 (MXVR_ALLOC_0 – MXVR_ALLOC_14)" on page 21-60
0xFFC0 2778 0xFFC0 277C 0xFFC0 2780 0xFFC0 2784 0xFFC0 2788 0xFFC0 278C 0xFFC0 2790 0xFFC0 2794	MXVR_SYNC_LCHAN_0 MXVR_SYNC_LCHAN_1 MXVR_SYNC_LCHAN_2 MXVR_SYNC_LCHAN_3 MXVR_SYNC_LCHAN_4 MXVR_SYNC_LCHAN_5 MXVR_SYNC_LCHAN_6 MXVR_SYNC_LCHAN_7	"MXVR Synchronous Logical Channel Assignment Registers (MXVR_SYNC_LCHAN_0 – MXVR_SYNC_LCHAN_7)" on page 21-62
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 Channel x Configuration Registers (MXVR_DMA0_CONFIG – MXVR_DMA7_CONFIG)" on page 21-64

Table B-28. MXVR Interface Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 279C 0xFFC0 27B0 0xFFC0 27C4 0xFFC0 27D8 0xFFC0 27EC 0xFFC0 2800 0xFFC0 2814 0xFFC0 2828	MXVR_DMA0_START_ADDR MXVR_DMA1_START_ADDR MXVR_DMA2_START_ADDR MXVR_DMA3_START_ADDR MXVR_DMA4_START_ADDR MXVR_DMA5_START_ADDR MXVR_DMA6_START_ADDR MXVR_DMA7_START_ADDR	"MXVR DMA Channel x Start Address Registers (MXVR_DMA0_START_ADDR – MXVR_DMA7_START_ADDR)" on page 21-73
0xFFC0 27A0 0xFFC0 27B4 0xFFC0 27C8 0xFFC0 27DC 0xFFC0 27F0 0xFFC0 2804 0xFFC0 2818 0xFFC0 282C	MXVR_DMA0_COUNT MXVR_DMA1_COUNT MXVR_DMA2_COUNT MXVR_DMA3_COUNT MXVR_DMA4_COUNT MXVR_DMA5_COUNT MXVR_DMA6_COUNT MXVR_DMA7_COUNT	"MXVR DMA Channel x Transfer Count Registers (MXVR_DMA0_COUNT – MXVR_DMA7_COUNT)" on page 21-75
0xFFC0 27A4 0xFFC0 27B8 0xFFC0 27CC 0xFFC0 27E0 0xFFC0 27F4 0xFFC0 2808 0xFFC0 281C 0xFFC0 2830	MXVR_DMA0_CURR_ADDR MXVR_DMA1_CURR_ADDR MXVR_DMA2_CURR_ADDR MXVR_DMA3_CURR_ADDR MXVR_DMA4_CURR_ADDR MXVR_DMA5_CURR_ADDR MXVR_DMA6_CURR_ADDR MXVR_DMA7_CURR_ADDR	"MXVR DMA Channel x Current Address Registers (MXVR_DMA0_CURR_ADDR – MXVR_DMA7_CURR_ADDR)" on page 21-75

Table B-28. MXVR Interface Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 27A8 0xFFC0 27BC 0xFFC0 27D0 0xFFC0 27E4 0xFFC0 27F8 0xFFC0 280C 0xFFC0 2820 0xFFC0 2834	MXVR_DMA0_CURR_COUNT MXVR_DMA1_CURR_COUNT MXVR_DMA2_CURR_COUNT MXVR_DMA3_CURR_COUNT MXVR_DMA4_CURR_COUNT MXVR_DMA5_CURR_COUNT MXVR_DMA6_CURR_COUNT MXVR_DMA7_CURR_COUNT	"MXVR DMA Channel x Current Transfer Count Registers (MXVR_DMA0_CURR_COUNT – MXVR_DMA7_CURR_COUNT)" on page 21-78
0xFFC0 2838	MXVR_AP_CTL	"MXVR Asynchronous Packet Control Register (MXVR_AP_CTL)" on page 21-80
0xFFC0 283C	MXVR_APRB_START_ADDR	"MXVR Asynchronous Packet Receive Buffer Start Address Register (MXVR_APRB_START_ADDR)" on page 21-83
0xFFC0 2840	MXVR_APRB_CURR_ADDR	"MXVR Asynchronous Packet Receive Buffer Current Address Register (MXVR_APRB_CURR_ADDR)" on page 21-84
0xFFC0 2844	MXVR_APTB_START_ADDR	"MXVR Asynchronous Packet Transmit Buffer Start Address Register (MXVR_APTB_START_ADDR)" on page 21-85
0xFFC0 2848	MXVR_APTB_CURR_ADDR	"MXVR Asynchronous Packet Transmit Buffer Current Address Register (MXVR_APTB_CURR_ADDR)" on page 21-86

Table B-28. MXVR Interface Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 284C	MXVR_CM_CTL	"MXVR Control Message Control Register (MXVR_CM_CTL)" on page 21-86
0xFFC0 2850	MXVR_CMRB_START_ADDR	"MXVR Control Message Receive Buffer Start Address Register (MXVR_CMRB_START_ADDR)" on page 21-89
0xFFC0 2854	MXVR_CMRB_CURR_ADDR	"MXVR Control Message Receive Buffer Current Address Register (MXVR_CMRB_CURR_ADDR)" on page 21-90
0xFFC0 2858	MXVR_CMTB_START_ADDR	"MXVR Control Message Transmit Buffer Start Address Register (MXVR_CMTB_START_ADDR)" on page 21-91
0xFFC0 285C	MXVR_CMTB_CURR_ADDR	"MXVR Control Message Transmit Buffer Current Address Register (MXVR_CMTB_CURR_ADDR)" on page 21-92
0xFFC0 2860	MXVR_RRDB_START_ADDR	"MXVR Remote Read Buffer Start Address Register (MXVR_RRDB_START_ADDR)" on page 21-93
0xFFC0 2864	MXVR_RRDB_CURR_ADDR	"MXVR Remote Read Buffer Current Address Register (MXVR_RRDB_CURR_ADDR)" on page 21-94
0xFFC0 2868 0xFFC0 2870	MXVR_PAT_DATA_0 MXVR_PAT_DATA_1	"MXVR Pattern Data Registers (MXVR_PAT_DATA_0, MXVR_PAT_DATA_1)" on page 21-95
0xFFC0 286C 0xFFC0 2874	MXVR_PAT_EN_0 MXVR_PAT_EN_1	"MXVR Pattern Enable Registers (MXVR_PAT_EN_0, MXVR_PAT_EN_1)" on page 21-96

Table B-28. MXVR Interface Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2878 0xFFC0 287C	MXVR_FRAME_CNT_0 MXVR_FRAME_CNT_1	"MXVR Frame Counter Registers (MXVR_FRAME_CNT_0, MXVR_FRAME_CNT_1)" on page 21-97
0xFFC0 2880 0xFFC0 2884 0xFFC0 2888 0xFFC0 288C 0xFFC0 2890 0xFFC0 2894 0xFFC0 2898 0xFFC0 289C 0xFFC0 28A0 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 (MXVR_ROUTING_0 – MXVR_ROUTING_14)" on page 21-98
0xFFC0 28BC	MXVR_PLL_CTL_1	"MXVR PLL Control Register 1 (MXVR_PLL_CTL_1)" on page 21-18
0xFFC0 28C0	MXVR_BLOCK_CNT	"MXVR Block Counter Register (MXVR_BLOCK_CNT)" on page 21-101
0xFFC0 28C4	MXVR_PLL_CTL_2	"MXVR PLL Control Register 2 (MXVR_PLL_CTL_2)" on page 21-20

CAN Registers

CAN registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in Table B-29 through Table B-32.

Table B-29. CAN Control and Configuration Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2A04 0xFFC0 2A44	CAN_MD1 CAN_MD2	"CAN_MDx Mailbox Direction Registers" on page 19-72
0xFFC0 2A08 0xFFC0 2A48	CAN_TRS1 CAN_TRS2	"CAN_TRSx Registers" on page 19-76
0xFFC0 2A0C 0xFFC0 2A4C	CAN_TRR1 CAN_TRR2	"CAN_TRRx Registers" on page 19-77
0xFFC0 2A10 0xFFC0 2A50	CAN_TA1 CAN_TA2	"CAN_TAx Registers" on page 19-79
0xFFC0 2A14 0xFFC0 2A54	CAN_AA1 CAN_AA2	"CAN_AAx Registers" on page 19-78
0xFFC0 2A18 0xFFC0 2A58	CAN_RMP1 CAN_RMP2	"CAN_RMPx Registers" on page 19-73
0xFFC0 2A1C 0xFFC0 2A5C	CAN_RML1 CAN_RML2	"CAN_RMLx Registers" on page 19-74
0xFFC0 2A20 0xFFC0 2A60	CAN_MBTIF1 CAN_MBTIF2	"CAN_MBTIFx Registers" on page 19-83
0xFFC0 2A24 0xFFC0 2A64	CAN_MBRIF1 CAN_MBRIF2	"CAN_MBRIFx Registers" on page 19-84
0xFFC0 2A28 0xFFC0 2A68	CAN_MBIM1 CAN_MBIM2	"CAN_MBIMx Registers" on page 19-82
0xFFC0 2A2C 0xFFC0 2A6C	CAN_RFH1 CAN_RFH2	"CAN_RFHx Registers" on page 19-80
0xFFC0 2A30 0xFFC0 2A70	CAN_OPSS1 CAN_OPSS2	"CAN_OPSSx Register" on page 19-75
0xFFC0 2A00 0xFFC0 2A40	CAN_MC1 CAN_MC2	"CAN_MCx Mailbox Configuration Registers" on page 19-71

Table B-29. CAN Control and Configuration Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2A80	CAN_CLOCK	"CAN_CLOCK Register" on page 19-47
0xFFC0 2A84	CAN_TIMING	"CAN_TIMING Register" on page 19-48
0xFFC0 2A88	CAN_DEBUG	"CAN_DEBUG Register" on page 19-47
0xFFC0 2A8C	CAN_STATUS	"CAN_STATUS Global CAN Status Register" on page 19-46
0xFFC0 2A90	CAN_CEC	"CAN_CEC Register" on page 19-87
0xFFC0 2A94	CAN_GIS	"CAN_GIS Global CAN Interrupt Status Register" on page 19-49
0xFFC0 2A98	CAN_GIM	"CAN_GIM Global CAN Interrupt Mask Register" on page 19-49
0xFFC0 2A9C	CAN_GIF	"CAN_GIF Global CAN Interrupt Flag Register" on page 19-50
0xFFC0 2AA0	CAN_CONTROL	"CAN_CONTROL Master Control Register" on page 19-45
0xFFC0 2AA4	CAN_INTR	"CAN_INTR Interrupt Pending Register" on page 19-48
0xFFC0 2AAC	CAN_MBTD	"CAN_MBTD Register" on page 19-80
0xFFC0 2AB0	CAN_EWR	"CAN_EWR Register" on page 19-87
0xFFC0 2AB4	CAN_ESR	"CAN_ESR Register" on page 19-87
0xFFC0 2AC4	CAN_UCCNT	"CAN_UCCNT Register" on page 19-86
0xFFC0 2AC8	CAN_UCRC	"CAN_UCRC Register" on page 19-86
0xFFC0 2ACC	CAN_UCCNF	"CAN_UCCNF Register" on page 19-85

Table B-30. CAN Mailbox Acceptance Mask Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2B00	CAN_AM00L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B04	CAN_AM00H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B08	CAN_AM01L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B0C	CAN_AM01H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B10	CAN_AM02L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B14	CAN_AM02H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B18	CAN_AM03L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B1C	CAN_AM03H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B20	CAN_AM04L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B24	CAN_AM04H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B28	CAN_AM05L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B2C	CAN_AM05H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B30	CAN_AM06L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B34	CAN_AM06H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B38	CAN_AM07L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50

Table B-30. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2B3C	CAN_AM07H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B40	CAN_AM08L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B44	CAN_AM08H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B48	CAN_AM09L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B4C	CAN_AM09H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B50	CAN_AM10L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B54	CAN_AM10H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B58	CAN_AM11L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B5C	CAN_AM11H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B60	CAN_AM12L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B64	CAN_AM12H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B68	CAN_AM13L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B6C	CAN_AM13H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B70	CAN_AM14L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B74	CAN_AM14H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50

Table B-30. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2B78	CAN_AM15L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B7C	CAN_AM15H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B80	CAN_AM16L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B84	CAN_AM16H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B88	CAN_AM17L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B8C	CAN_AM17H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B90	CAN_AM18L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B94	CAN_AM18H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B98	CAN_AM19L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2B9C	CAN_AM19H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BA0	CAN_AM20L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BA4	CAN_AM20H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BA8	CAN_AM21L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BAC	CAN_AM21H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BB0	CAN_AM22L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50

Table B-30. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2BB4	CAN_AM22H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BB8	CAN_AM23L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BBC	CAN_AM23H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BC0	CAN_AM24L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BC4	CAN_AM24H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BC8	CAN_AM25L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BCC	CAN_AM25H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BD0	CAN_AM26L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BD4	CAN_AM26H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BD8	CAN_AM27L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BDC	CAN_AM27H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BE0	CAN_AM28L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BE4	CAN_AM28H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BE8	CAN_AM29L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BEC	CAN_AM29H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50

Table B-30. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2BF0	CAN_AM30L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BF4	CAN_AM30H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BF8	CAN_AM31L	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50
0xFFC0 2BFC	CAN_AM31H	"CAN_AMxx Mailbox Acceptance Registers" on page 19-50

Since each CAN mailbox has an identical MMR set, with fixed offsets from the base address associated with that mailbox, it is convenient to view the MMR information as provided in Table B-31 and Table B-32. Table B-31 identifies the base address of each CAN mailbox, as well as the register prefix that identifies mailbox. Table B-32 then lists the register suffix and provides its offset from the base address.

As an example, the CAN mailbox 2 length register is called `CAN_MB02_LENGTH`, and its address is 0xFFC0 2C50. Likewise, the CAN mailbox 17 timestamp register is called `CAN_MB17_TIMESTAMP`, and its address is 0xFFC0 2E34.

Table B-31. CAN Mailbox Base Addresses

Mailbox Identifier	MMR Base Address	Register Prefix
0	0xFFC0 2C00	CAN_MB00_
1	0xFFC0 2C20	CAN_MB01_
2	0xFFC0 2C40	CAN_MB02_
3	0xFFC0 2C60	CAN_MB03_
4	0xFFC0 2C80	CAN_MB04_
5	0xFFC0 2CA0	CAN_MB05_

Table B-31. CAN Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
6	0xFFC0 2CC0	CAN_MB06_
7	0xFFC0 2CE0	CAN_MB07_
8	0xFFC0 2D00	CAN_MB08_
9	0xFFC0 2D20	CAN_MB09_
10	0xFFC0 2D40	CAN_MB10_
11	0xFFC0 2D60	CAN_MB11_
12	0xFFC0 2D80	CAN_MB12_
13	0xFFC0 2DA0	CAN_MB13_
14	0xFFC0 2DC0	CAN_MB14_
15	0xFFC0 2DE0	CAN_MB15_
16	0xFFC0 2E00	CAN_MB16_
17	0xFFC0 2E20	CAN_MB17_
18	0xFFC0 2E40	CAN_MB18_
19	0xFFC0 2E60	CAN_MB19_
20	0xFFC0 2E80	CAN_MB20_
21	0xFFC0 2EA0	CAN_MB21_
22	0xFFC0 2EC0	CAN_MB22_
23	0xFFC0 2EE0	CAN_MB23_
24	0xFFC0 2F00	CAN_MB24_
25	0xFFC0 2F20	CAN_MB25_
26	0xFFC0 2F40	CAN_MB26_
27	0xFFC0 2F60	CAN_MB27_
28	0xFFC0 2F80	CAN_MB28_
29	0xFFC0 2FA0	CAN_MB29_
30	0xFFC0 2FC0	CAN_MB30_
31	0xFFC0 2FE0	CAN_MB31_

Table B-32. CAN Mailbox Register Suffix and Offset

Register Suffix	Offset From Base	See Page
DATA0	0x00	"CAN_MBxx_DATAx Registers" on page 19-63
DATA1	0x04	"CAN_MBxx_DATAx Registers" on page 19-63
DATA2	0x08	"CAN_MBxx_DATAx Registers" on page 19-63
DATA3	0x0C	"CAN_MBxx_DATAx Registers" on page 19-63
LENGTH	0x10	"CAN_MBxx_LENGTH Registers" on page 19-61
TIME-STAMP	0x14	"CAN_MBxx_TIMESTAMP Registers" on page 19-59
ID0	0x18	"CAN_MBxx_ID0 Registers" on page 19-57
ID1	0x1C	"CAN_MBxx_ID1 Registers" on page 19-55

C TEST FEATURES

This chapter discusses the test features of the processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a Boundary-Scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a Boundary-Scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table)
- A TAP controller that controls all sequencing of events through the test registers
- An Instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table C-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure C-1 shows the state diagram for the TAP controller.

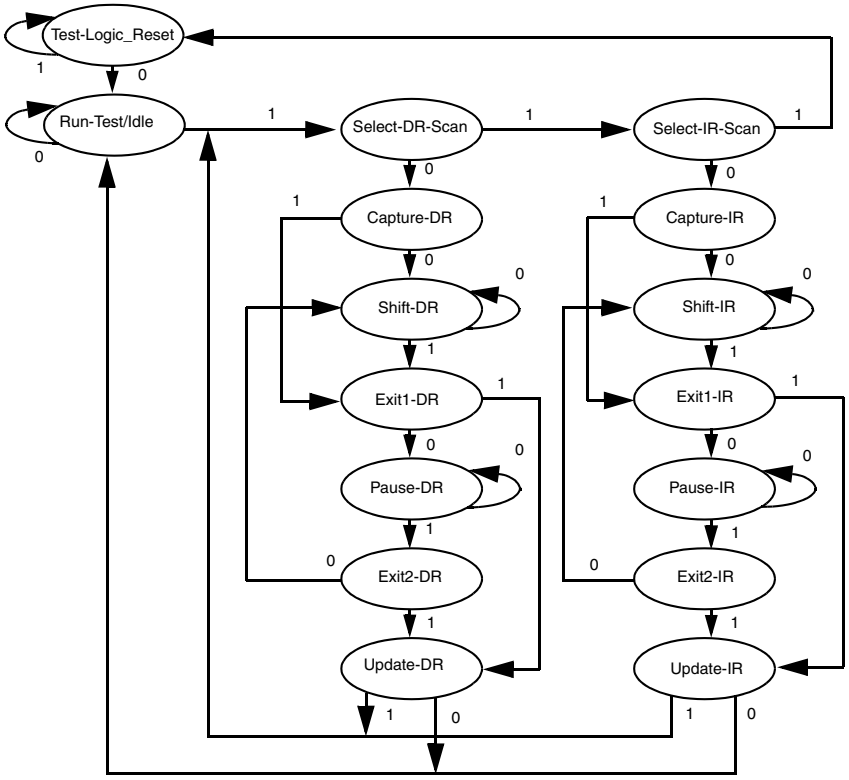


Figure C-1. TAP Controller State Diagram

Note:

- The TAP controller enters the Test-Logic-Reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the Test-Logic-Reset state when TRST is asynchronously asserted.
- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The Instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The Instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table C-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table C-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure C-2 shows the instruction bit scan ordering for the paths shown in Table C-2.

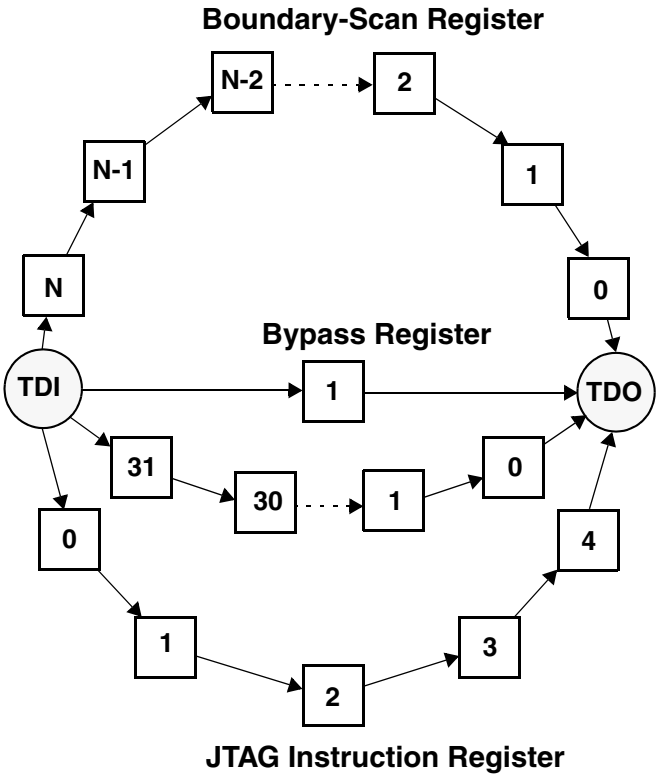


Figure C-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The EXTEST instruction selects the Boundary-Scan register to be connected between the TDI and TDO pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

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.

D NUMERIC FORMATS

Blackfin family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two's-complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the Blackfin family are in two's-complement format. Signed-magnitude, one's-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional

The Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure D-1. Note in two's-complement format, the sign bit has a negative weight.

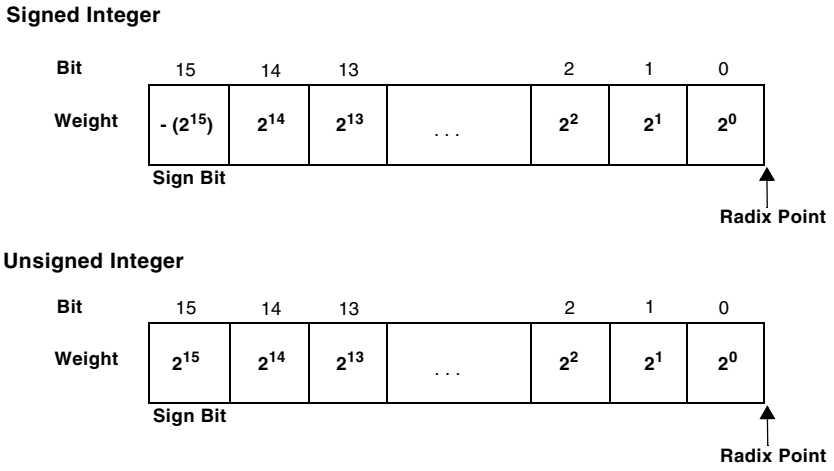


Figure D-1. Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure D-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N – 1.

The notation used to describe a format consists of two numbers separated by a period (.); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure D-2 is 13.3.

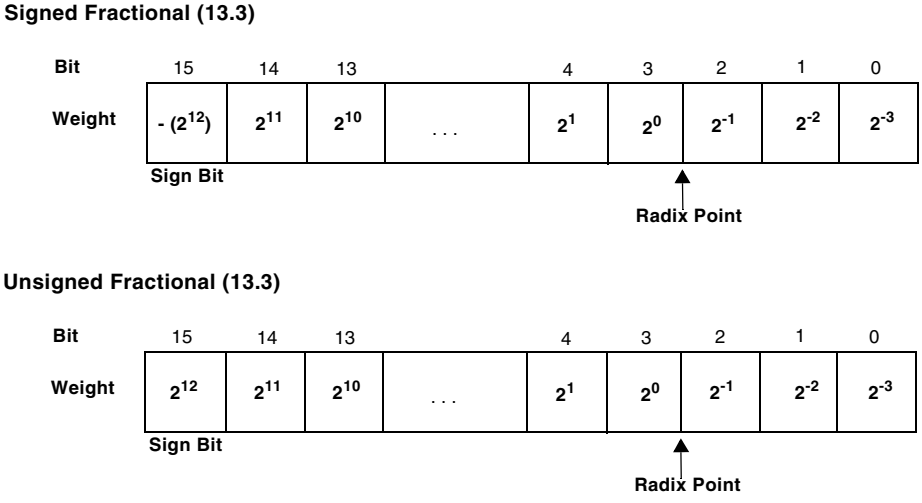


Figure D-2. Example of Fractional Format

Table D-1 shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Table D-1. Fractional Formats and Their Ranges

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	–1.0	0.000030517578125
2.14	2	14	1.999938964843750	–2.0	0.000061035156250
3.13	3	13	3.999877929687500	–4.0	0.000122070312500
4.12	4	12	7.999755859375000	–8.0	0.000244140625000
5.11	5	11	15.999511718750000	–16.0	0.000488281250000
6.10	6	10	31.999023437500000	–32.0	0.000976562500000
7.9	7	9	63.998046875000000	–64.0	0.001953125000000
8.8	8	8	127.996093750000000	–128.0	0.003906250000000
9.7	9	7	255.992187500000000	–256.0	0.007812500000000
10.6	10	6	511.984375000000000	–512.0	0.015625000000000
11.5	11	5	1023.968750000000000	–1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	–2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	–4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	–8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

Binary Multiplication

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The Blackfin family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in Figure D-3. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

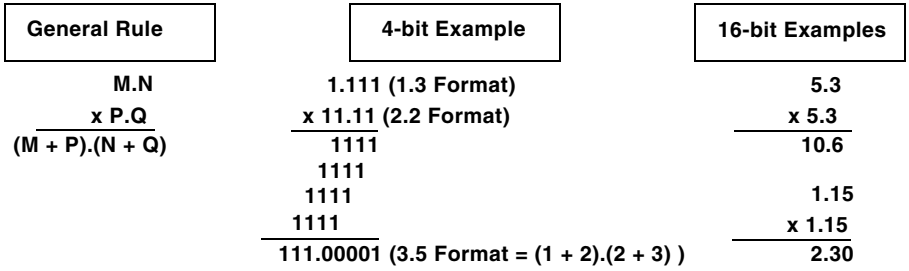


Figure D-3. Format of Multiplier Result

Fractional Mode And Integer Mode

A product of 2 two's-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The Blackfin family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to

the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

For more information about data formats, see the data formats listed in Table 2-2 on page 2-11.

Block Floating-point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See Figure D-4. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

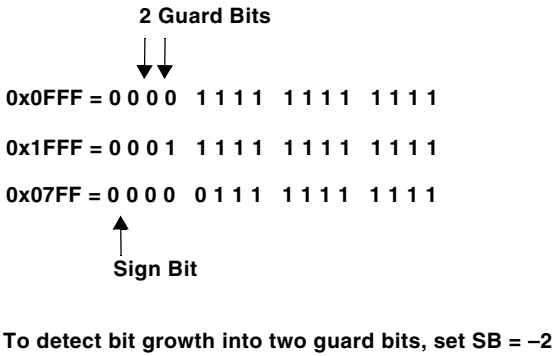


Figure D-4. Data With Guard Bits

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

Figure D-5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the `SIGNBITS` instruction is `SB` and `SB` is used as an argument in the `EXPADJ` instruction (see *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for the usage and syntax of these instructions). Initially, the value of `SB` is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the `EXPADJ` instruction, which counts the number of redundant sign bits and adjusts `SB` if the number of redundant sign bits is less than two. In this example, `SB` = +1 after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.
- If `SB` were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

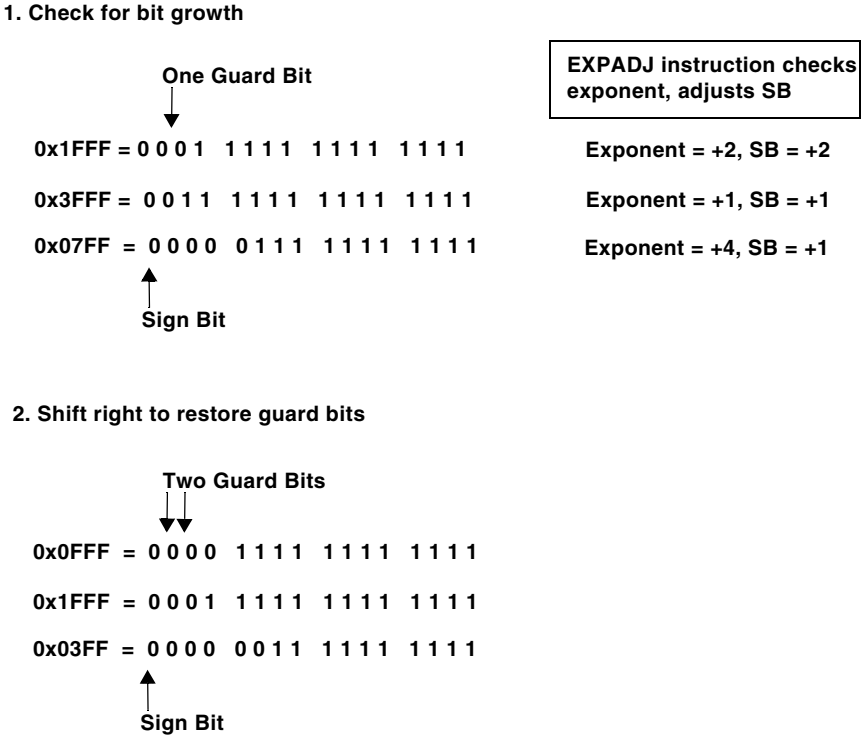


Figure D-5. Block Floating-point Adjustment

I INDEX

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z